LATCH-BASED PERFORMANCE OPTIMIZATION FOR FPGAs

by

Xiao Teng

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of ECE
University of Toronto

Copyright © 2012 by Xiao Teng

Abstract

Latch-Based Performance Optimization for FPGAs

Xiao Teng

Master of Applied Science

Graduate Department of ECE

University of Toronto

2012

We explore using pulsed latches for timing optimization – a first in the academic FPGA community. Pulsed latches are transparent latches driven by a clock with a non-standard (i.e. not 50%) duty cycle. As latches are already present on commercial FPGAs, their use for timing optimization can avoid the power or area drawbacks associated with other techniques such as clock skew and retiming. We propose algorithms that automatically replace certain flip-flops with latches for performance gains. Under conservative short path or minimum delay assumptions, our latch-based optimization, operating on already routed designs, provides all the benefit of clock skew in most cases and increases performance by 9%, on average, essentially for “free”. We show that short paths greatly hinder the ability of using pulsed latches, and further improvements in performance are possible by increasing the delay of certain short paths.

Acknowledgements

The research was good. The people were extraordinary. I'd like to thank:

1. Professor Jason Anderson for his guidance.
2. The people of PT477 and PT392 for creating a fun and diverse learning environment.
3. To mom for the support.

Contents

1	Introduction	1
1.1	Contributions	4
1.2	Thesis Organization	5
2	Related Work	6
2.1	FPGA Architecture	6
2.2	FPGA CAD Flow	9
2.2.1	Placement	10
2.2.2	Routing	10
2.2.3	Timing Analysis	12
2.3	Clock Skew	16
2.4	Retiming	18
3	Level-Sensitive Latches	21
3.1	Latch Basics	21
3.2	Timing Constraints	24
3.3	Simplifying Latch Timing Constraints	27
3.4	Prior Work	29
4	Graph-Theoretic Timing Optimization	30
4.1	Preliminaries	30

4.2	Calculating The Optimal Clock Period	31
4.2.1	Long Path Constraints	31
4.2.2	Short Path and Special Constraints	33
4.2.3	Input-Output Paths	34
4.3	Howard’s Algorithm	34
4.3.1	Value Determination	35
4.3.2	Policy Improvement	38
5	Latch-Based Timing Optimization	40
5.1	Post Place and Route Latch Insertion	40
5.2	Iterative Improvement	42
5.3	Optimality	46
5.4	Experimental Study	48
6	Delay Padding	53
6.1	Pulse Width Selection	54
6.2	Short Path Identification	55
6.3	Delay Padding Strategies	58
6.4	Intra-CLB Paths	60
6.5	Experimental Study	61
7	Conclusions and Future Work	67
	Bibliography	69

List of Tables

2.1	Summary of flip-flop timing parameters.	13
3.1	Summary of latch timing parameters	25
5.1	Achievable Clock period (ns) using flip-flops without any time borrowing (Critical Path), optimal clock period without considering short path constraints (PL_{opt}), pulsed latches heuristic (PL_{heur}), iterative improvement with pulsed latches (PL_{iter}), and clock skew (CS) subject to different minimum delay assumptions.	52
6.1	Clock period reduction under different pulse width offset assumptions by delay padding.	63
6.2	Comparing the number short paths requiring delay padding with and without the use of using flip-flops.	66

List of Figures

1.1	Illustration showing how varying the pulse width can fix hold-time violations	4
2.1	Basic Logic Element (BLE)	7
2.2	Conventional FPGA architecture	8
2.3	FPGA CAD flow	9
2.4	PathFinder algorithm	11
2.5	Slack computation	14
2.6	Clock skew benefits and potential hazards	16
2.7	Retiming example	19
2.8	A retiming move that does not have an equivalent initial state	20
3.1	Latch basics	22
3.2	Level-sensitive latch timing parameters	23
3.3	The advantage of pulsed latches	24
4.1	Sample circuit fragment and its graph representation	32
4.2	Detecting critical I/O paths	35
4.3	Howard's Algorithm	36
5.1	Iterative improvement example	45
5.2	Contrasting the greedy and iterative pulse width selection approaches	47
5.3	Illustrating the advantage of clock skew over pulse latches and its limitations	51

6.1	Delay padding in relation to rest of CAD flow	54
6.2	Different route rip-up scenarios for minimally disruptive delay padding .	58
6.3	Delay padding flowchart	59
6.4	Performance gains in relation to additional wirelength necessary to fix short path violations	64
7.1	Comparing the benefits of pulsed latches with and without using cycle-slacks in VPR	69

Chapter 1

Introduction

FPGAs are programmable digital circuits that quickly allow the implementation a wide array of digital designs. The advancement of process technology, architectural and computer-aided design (CAD) [8] research has allowed FPGAs to be a viable design platform for an ever-increasing number of applications [11,69,76,81] since their introduction over 20 years ago [19]. Unlike application-specific integrated circuits (ASICs), FPGAs allow rapid design prototyping, incremental design debugging, and also avoid the high non-recurring engineering costs.

Unfortunately, the advantages of programmability come at a price: area, performance, and power consumption. A study [29] showed that FPGA designs require $20\text{-}40\times$ more area, $12\times$ more dynamic power, and are $3\text{-}4\times$ slower than their equivalent ASIC implementation. Since process technology advancement benefits both ASICs and FPGAs, it is clear that novel architectural and CAD techniques for FPGAs are necessary to close the gap.

Our work explores how FPGA designs can be made to run faster. One well-known approach, clock skew scheduling, intentionally delays some clocks to certain flip-flops to steal time from subsequent combinational stages [18]. For example, for a combinational logic path from a flip-flop j to flip-flop i , clock skew scheduling may delay the arrival of the clock signal to i , thereby allowing more time for a logic signal to arrive at i 's

input. Research [7,48,62] has shown that only a few skewed clocks are necessary to obtain appreciable improvements in circuit speed. Unfortunately, clocks comprise 20-39% of dynamic power consumption in commercial FPGAs [14,56]. Since FPGAs already consume more dynamic power than ASICs, it is clear that for FPGAs to remain competitive with ASICs, it would be desirable to improve circuit performance *without* using extra clocks. Our work explores how FPGA designs can be made to run faster. One well-known approach, clock skew scheduling, intentionally delays some clocks to certain flip-flops to steal time from subsequent combinational stages [18]. For example, for a combinational logic path from a flip-flop j to flip-flop i , clock skew scheduling may delay the arrival of the clock signal to i , thereby allowing more time for a logic signal to arrive at i 's input. Research [7,48,62] has shown that only a few skewed clocks are necessary to obtain appreciable improvements in circuit speed. Unfortunately, clocks comprise 20-39% of dynamic power consumption in commercial FPGAs [14,56]. Since FPGAs already consume more dynamic power than ASICs, it is clear that for FPGAs to remain competitive with ASICs, it would be desirable to improve circuit performance *without* using extra clocks.

Another approach to borrowing time is *retiming*, which physically relocates flip-flops across combination logic to balance the delays between combinational stages. Although extra clock lines are not required to borrow time, the practical usage of retiming is limited due to its impact on the verification methodology, i.e., equivalence checking and functional simulation [48]. Retiming can change the number of flip-flops in a design, for example, in the case of moving a flip-flop “upstream” from the output of a multi-input logic gate to its inputs. As such, retiming may increase circuit area, and make it difficult for a designer to verify functionality or to correlate the retimed design with the original RTL specification.

Our approach involves using a mix of level-sensitive latches and regular flip-flops. By doing so, we can avoid the power barrier associated with using multiple clocks, and the netlist modifications required for retiming. Level-sensitive latches achieve time borrowing by providing a window of time in which signals can freely pass through. We say a latch

is *transparent* during this window of time. Consider again a combinational path from a flip-flop j to a latch i . The maximum allowable delay for the path can extend beyond the clock period. Specifically, a transition launched from j need not settle on i 's input by the next rising clock edge. It may settle after the clock edge during the time window when i is transparent. The downside is that timing analysis using latches is more difficult because transparency allows critical long (max delay) and short (min delay) paths to extend across multiple combinational stages, unlike standard timing analysis using flip-flops. Furthermore, a larger transparency window may allow long paths to borrow more time, but also make the circuit more susceptible to hold-time violations. As the presence of a single violation can render a design inoperable, special attention must be paid to satisfying short paths. Using *pulsed latches* driven by a clock with a non-standard duty cycle or *pulse width* (i.e. not 50%), is one method of reducing the effects of short paths plaguing conventional latch-based circuits, while allowing time borrowing for long paths. This is a viable option as commercial FPGAs can generate clocks with different duty cycles, as well as allow the sequential elements in Combinational Logic Blocks (CLBs) to be used as either flip-flops or latches [77, 78]. That is, commercial FPGAs already contain the necessary hardware functionality to support pulsed latch-based timing optimization, but to the best of our knowledge, no prior academic work [36] has explored the pulsed latch concept for FPGAs.

The advantage of using pulsed latches is shown in Fig. 1.1. Solid and dashed lines represent long and short combinational paths, respectively, between latch L_2 , FF_1 , and FF_3 . If a pulse width of 3 time units is used, it is possible that two signals launched on two different clock cycles can arrive at one flip-flop (FF_3) at the same time, which clearly is invalid. The cause of this problem is the short path signal launched from FF_1 arriving at L_2 when it is still transparent - a hold-time or short path violation. One way to fix this violation is to reduce the pulse width to 2. As a result, the short path would not arrive at L_2 when it is transparent and launch in the next cycle instead. Naturally, the

Figure 1.1: Illustration showing how varying the pulse width can fix hold-time violations

use of larger pulse widths permit more time borrowing between adjacent combinational stages, allowing larger improvements in timing performance; however, larger pulse widths are more likely to create hold-time violations.

1.1 Contributions

The major contributions of our work are:

- We are the first in academia [36] to explore using pulsed latches for timing optimization in FPGAs, which was first published in [68].
- Our algorithms can selectively insert latches into already-routed flip-flop-based designs for improved timing performance without extra clocks or logic.
- Our experiments show that all of the performance improvements achieved by clock skew can also be attained with our optimization with a *single* clock for most benchmarks.

- We explore different methods of increasing the delay of short paths for further performance improvement, each with benefits and drawbacks.
- We devise a heuristic that forces the use of flip-flops in certain cases to avoid fixing the majority of short path violations caused by the transparency nature of latches.

1.2 Thesis Organization

The remainder of this thesis is broken down into several chapters. Chapter 2 provides the necessary background on FPGA architecture and CAD flow needed to understand how and where our optimizations fit. This chapter also reviews two popular time borrowing methods: clock skew and retiming. Chapter 3 discusses the basics of a level-sensitive latch, its timing constraints, and how they can be transformed so that well-known optimization techniques can be applied. Chapter 4 discusses how timing optimization using level-sensitive latches can be formulated and optimized in a graph-theoretic manner. Chapter 5 discusses two “free” optimizations that automatically insert level-sensitive latches into conventional flip-flop based circuits for performance improvements. Results presented in Chapter 5 showed that short path constraints can severely limit the possible gains with latches. To alleviate this problem, Chapter 6 discusses two different strategies to *increase* the delay of certain short paths so that further performance improvements are possible. Finally, we conclude and provide insight into future directions of our work in Chapter 7.

Chapter 2

Related Work

This chapter reviews prior work necessary for understanding latch-based timing optimization. Section 2.1 gives an introduction to FPGA architecture and points out the different sources of delay. Section 2.2 discusses the sections of the FPGA CAD flow relevant to our work. Then, we move onto different methods of implementing time borrowing. We review two approaches that have already been explored in the FPGA community: clock skew in Section 2.3 and retiming in Section 2.4.

2.1 FPGA Architecture

The fundamental unit of logic in an FPGA is a lookup-table (LUT). A LUT with k inputs (k-LUT) is essentially a 2^k -to-1 configurable multiplexer with static RAM (SRAM) bits driving its inputs, as shown in Fig. 2.1. This configuration allows a k-LUT to implement any k-input function by setting the SRAM bits. A flip-flop and a 2-to-1 multiplexer is bundled together with the k-LUT allow implementation of sequential circuits. This bundle is known as a *Basic Logic Element* (BLE). A larger LUT allows more logic to be implemented per LUT, and usually leads to a lower number of LUTs and routing resources on the critical path. However, larger LUTs are slower and area requirements grow exponentially [49] with the number of inputs. One method to accommodate more

Figure 2.1: Basic Logic Element (BLE)

logic is to cluster several BLEs that use reasonably-sized LUTs into one logic block. This is known as a *Configurable Logic Block* (CLB). CLBs provide local interconnect that allow potential fan-in and fan-out logic to remain within the same CLB, giving the option for short connections between logic. Although it would be beneficial to use local fast connections for connecting all BLEs in an FPGA, increasing the number of BLEs in a cluster requires more area for logic and interconnect, which also leads to longer connections between CLBs. Ahmed and Rose [1] experimentally showed that using LUTs with 4 to 6 inputs with each CLB containing 4 to 10 BLEs resulted in the best area-delay tradeoff.

Fig. 2.2 gives an overview of the island-style FPGA architecture that is most well-known today. Because non-trivial designs cannot fit into a single CLB and would require multiple CLBs, connectivity between CLBs is achieved by the FPGA's routing architecture. A routing architecture contains *routing segments* that provide the necessary connectivity between CLBs. A CLB uses *programmable switches* to connect to adjacent routing segments for external connectivity. Programmable switches are also used inside *switch blocks* to connect incoming and outgoing routing segments. They are represented by the dashed lines inside the box.

Path delays are typically dominated by routing delays in FPGAs [57]. Although

Figure 2.2: Conventional FPGA architecture

Fig. 2.2 only depicts routing segments that span one CLB, wires that extend span multiple CLBs have been investigated [5]. Using longer wires require fewer programmable switches, which leads to less area occupied by switches and faster paths between CLBs. However, longer wires can potentially be wasteful for short connections that do not fully utilize the full length of the wire. The use of different programmable switches for speed and area tradeoffs have also been investigated [5, 57]. Two examples are pass transistors and tri-state buffers. Pass transistors require less area than buffers, but incur more delay.

Each of the routing architecture components contribute to the delay of connections between logic. Different routing architectures lead to different area-delay tradeoffs. Our approach attempts to mitigate the impact of logic and routing delay by allowing time to be shared across combinational stages, leading to better circuit performance without impacting area.

2.2 FPGA CAD Flow

The FPGA CAD flow is responsible for interpreting RTL, such as VHDL or Verilog, and mapping it to a given FPGA architecture with area, timing, or power optimization objectives. Fig. 2.3 shows the relationship between different stages of the CAD flow. The stages highlighted in grey in Fig. 2.3 will be described. Specifically, we will introduce placement in Section 2.2.1, routing in Section 2.2.2, and static timing analysis (STA) in Section 2.2.3. VPR 5.0 [39] does not include an optimization stage after routing, but this is where our initial latch-based optimizations are incorporated and will be discussed in later chapters.

Figure 2.3: FPGA CAD flow

2.2.1 Placement

Placement is responsible for mapping every CLB to a valid location on an FPGA. Since it is desirable to maximize circuit performance and minimize power, this implies the objective for placement is to minimize wire usage by placing connected CLBs closer together. However, bringing a pair of CLBs closer together most likely will widen their distance to other CLBs. Such potentially conflicting objectives make finding “good” placements a challenging task. Finding the optimal placement is almost impossible as no known polynomial-time algorithm exists. Therefore, good heuristics are necessary to find good solutions in a reasonable amount of time. Two popular approaches are used to solve the placement problem:

1. Analytical placers [6,26,73] formulate the placement problem as a quadratic program with the objective of minimizing the total wire usage. The results in an initial placement that has CLBs closely clustered together, with many overlapping CLBs. Clearly, this is an illegal placement and a subsequent spreading stage is necessary to ensure a valid final placement.
2. Iterative improvement placers [4,25,55], specifically simulated-annealing, start with an initial placement and incrementally adjust the placement by swapping CLBs based on cost functions, which model the optimization objective. VPR 5.0 [39], which we modify in this work, uses simulated annealing for its placement stage.

2.2.2 Routing

Routing is responsible for allocating routing resources to nets to form the necessary connections between CLBs. As shorter routes between pins would lead to better circuit performance, it may appear that a simple application of a shortest path algorithm for each net would suffice. Such a solution would not work because certain routing segments may be overused or *congested*, i.e. multiple nets driving a single routing segment. Since

multiple nets may contend for a single routing segment, it is almost certain that some nets will need a routing resource more than others to better satisfy some global optimization goal. PathFinder [41] is an FPGA routing algorithm that allow nets to *negotiate* amongst themselves towards a global optimization goal.

Figure 2.4: PathFinder algorithm

The main idea of PathFinder can be depicted by a simple flow chart as shown in Fig. 2.4. PathFinder permits routing solutions that overuse certain routing segments. Costs of overused routing segments, or *overflow costs* are increased to discourage such illegal solutions. The overflow costs are incorporated into future iterations. Given sufficient routing resources, some nets will opt to avoid segments with high overflow costs, thereby alleviating congestion. PathFinder terminates once a legal routing solution has been found.

PathFinder describes a flow that will eventually generate a legal routing solution. However, it still needs some way to route all nets. One method used by VPR 5.0 [39], is *maze expansion* [32]. Maze expansion is responsible for finding the set of routing segments that connect the source and target of a net with minimal cost. Costs may model a multitude of metrics such as delay, wire usage, and routing segment overuse.

The costing mechanism operates on the routing graph model, $G(V, E)$, representing the FPGA architecture. The vertices V model logic pins and wire segments, while edges E model the connectivity between such resources.

Maze expansion starts from the source and finds the minimal cost to the target by iteratively visiting adjacent nodes until the target is found. The source node is inserted into a priority queue with a starting cost to start the search. The priority queue dequeues or visits nodes with minimal cost first. When a node is dequeued, its adjacent nodes are labeled with a cost and inserted into the priority queue to further propagate the search. This process repeats until the target node is visited. The actual routing segments used to reach the target are known once a backtrace starting from the target node to the source node completes.

2.2.3 Timing Analysis

One of the major goals in synchronous circuit design is to maximize the speed of a circuit. Speed is measured by the frequency of the clock driving the sequential elements. The role of timing analysis is to characterize the performance of a synchronous circuit by determining the clock period that it can correctly operate at. As typical sequential designs use flip-flops, correct functionality is governed by the *setup-time* and *hold-time* constraints¹. The setup-time constraint ensures that no signal arrives at its destined flip-flop *after* the clock event, i.e. positive or negative edge of the clock. Specifically, every signal that starts at some flip-flop j connected to a flip-flop i must arrive i 's input within a single clock period, P . This can be succinctly summarized by the following constraint:

$$T_{cq} + CD_{ji} + T_{su} \leq P, \forall j \rightarrow i \quad (2.1)$$

T_{cq} , or *clock-to-Q* time, accounts for the lag time between the output of a flip-flop

¹They are also known as the *long path* and *short path* constraints, respectively.

reacting to its input after a clock event. CD_{ji} is the maximum combinational delay of any path starting and ending at flip-flops j and i , respectively. The setup-time, T_{su} , is a parameter of a flip-flop and represents a window of time before the subsequent clock event where the flip-flop's data input signal must remain stable for correct circuit functionality. Synchronous sequential circuits must also obey the hold-time constraint. It ensures that a signal does not arrive *too* early at its destination. Consider once again a flip-flop j connected to another flip-flop i through a network of combinational logic. It is possible that the data at j 's output can reach i so quickly that the data at i 's output gets corrupted. Satisfying the following inequality will prevent such a situation from occurring:

$$T_{cq} + cd_{ji} \geq T_h, \forall j \rightarrow i \quad (2.2)$$

cd_{ji} represents the delay of the fastest combinational path between flip-flops j and i . T_h , also known as the hold-time, is the minimum amount of time data at flip-flop i should be stable after a clock event. Although optimizing the circuit for performance by reducing P is one of the major objectives of circuit design, failure to meet all hold-time constraints would result in a circuit that will not function with *any* value of P . Table 2.1 gives a summary of the timing parameters described in this section.

T_{cq}	clock-to-Q delay
cd_{ji}, CD_{ji}	short and long $j \rightarrow i$ combinational path delay from flip-flop j to flip-flop i
P	clock period
T_{su}	setup-time
T_h	hold-time

Table 2.1: Summary of flip-flop timing parameters.

Static timing analysis (STA) provides a fast, input-independent method of calculating P by identifying the longest combinational path, or the *critical path* in the circuit. STA represents the combinational logic network as a graph with nodes representing logic gate

However, it would be beneficial to be aware of near-critical paths because any one of them may become the new critical path, if neglected. Driving timing optimization in each stage of the CAD flow with connection *slacks* [21] can alleviate this problem. The slack of a pin-to-pin connection gives the amount of delay that may be added to the connection before it participates on the critical path. Calculating the slack also requires computing the required time, $T_{required}(i)$, of every node. It represents the latest acceptable arrival time of any signal without increasing the critical path. Computation of $T_{required}(i)$ begins by setting $T_{required}(i)$ of every sink to the value P . The remaining nodes are visited in a backwards topological order using the following formula:

$$T_{required}(i) = \min_{\forall j \in fanout(i)} (T_{required}(j) - delay_{i,j}) \quad (2.4)$$

Fig. 2.5(c) gives an example of $T_{required}(i)$ computation starting from the sinks, with required times inscribed in each node. Given $T_{arrival}(i)$ and $T_{required}(i)$, Computing the slack of each connection, as shown in Fig. 2.5(d), can be done so using equation 2.5:

$$Slack_{i,j} = T_{required}(j) - T_{arrival}(i) - delay_{i,j} \quad (2.5)$$

The conventional FPGA CAD flow uses connection slacks by mapping them to a floating point number between 0 and 1 to signify the relative importance of a connection from a timing perspective. This is known as the *criticality* metric, as shown below:

$$Crit(i,j) = 1.0 - \frac{Slack(i,j)}{P} \quad (2.6)$$

For example, connections with no slack are on the critical path, and have a criticality of 1, while non-critical connections are assigned a value less than 1.

2.3 Clock Skew

Since the timing of logic signals depends on the global clock(s), it is important that clock distribution is done in a reliable and efficient manner. Unfortunately, real signals take time to travel from point to point and clock signals are no different. Therefore, it is possible that clocks can arrive at different sequential elements at different times. This is known as *clock skew*.

One way to account for clock skew is to minimize it in the clock network [27, 61, 71]. However, Fishburn [18] recognized that clock skew can be used as a manageable resource and help reduce the clock period. To illustrate this, consider Fig. 2.6. Without any time borrowing, the critical path is 8 ns from FF_1 to FF_2 . Using a clock period of 6 ns would result in a violation, as shown in the timing diagram of Fig. 2.6(a). However, if the clock to FF_2 can be intentionally delayed by 2 ns, a 6 ns clock period can satisfy the long path (solid line) between the two flip-flops, as shown in Fig. 2.6(b).

Figure 2.6: Clock skew benefits and potential hazards

The ability to borrow time can be modeled by modifying the setup-time constraint (2.1) to include additional terms:

$$D_j + T_{cq} + T_{su} + CD_{ji} \leq D_i + P, \forall j \rightarrow i \quad (2.7)$$

D_j and D_i represent delays on the clock arrival times driving flip-flops j and i ,

respectively. As the example in Fig. 2.6 showed, D_j and D_i give combinational stages the ability to borrow time from subsequent stages, leading to a clock period unattainable without time borrowing.

Hold-time violations still exist. The dashed line in the timing diagram of Fig 2.6(b) shows that the data stored at FF_1 can change the data stored at FF_2 's input before FF_2 's clock event occurs. To ensure this does not occur, modifying inequality (2.2) to also include D_i and D_j results in:

$$D_j + T_{cq} + cd_{ji} \geq D_i + T_h, \forall j \rightarrow i \quad (2.8)$$

Unlike conventional static timing analysis, D_i and D_j provide an additional dimension of freedom to reducing the clock period. This leads to a more complex optimization problem. Initial approaches applied linear programming [18] or graph algorithms [15] directly on (2.7) and (2.8) to find the optimal clock period. The downside to this approach is that there are no constraints on the possible range or granularity of D_j and D_i . Thus, achieving the optimal P may require many unique skews, which can be prohibitively expensive to implement if each unique skew corresponds to a separate clock signal. Studies have determined that a single clock signal can be responsible for 20% of the total dynamic power dissipation [14, 56]. Thus, much work has been devoted to finding efficient methods of stealing time with a finite number of clocks [7, 37, 45, 48]. Specifically for FPGAs, Singh and Brown showed that 4 shifted clock lines provides over a 20% improvement in circuit speed [62].

Other work [17, 79] involving FPGAs has focused on the use of programmable delay elements (*PDEs*) to purposely delay clock signals. PDEs allow for fine-grain control of skews, which may make direct implementation of inequalities (2.7) and (2.8) possible. The work presented in [79] used PDEs on the clock tree, whereas the PDEs were inserted into FPGA logic elements in [17]. Both methods incur a hardware penalty and require additional architectural considerations.

2.4 Retiming

Another approach to borrowing time is *retiming*. Retiming physically relocates flip-flops or latches across combinational logic to balance the delays between combinational stages. Retiming can reduce the clock period, area, or both for a given circuit, without altering its functionality. Sequential elements can move *backward* or *forward*. A forward push of a flip-flop gives the combinational stage feeding into the flip-flop *more* time to complete, whereas a backwards move has the opposite effect. To illustrate this, consider the example given in Fig. 2.7. Combinational delays are inscribed inside the logic gates and assume that wire delays are zero. Also, assume that the black boxes are flip-flops. Fig. 2.7(a) highlights the critical path, 7 time units, without any retiming. If we push flip-flops A, B, and C forward, as shown in Fig. 2.7(b), we can reduce the critical path to 4 time units and reduce flip-flop usage by 2. This configuration yields minimal flip-flop usage, but the clock period can be further reduced. Fig. 2.7(c) gives a configuration that doesn't reduce the number of flip-flops, but results in a critical path of 3 time units by moving FF_A , FF_B , FF_C forward and FF_D backward.

Retiming was first introduced by Leiserson and Saxe [35]. Using a graph-theoretic approach, they provided algorithms that minimized the clock period or number of flip-flops. Their initial work has been extended in multiple directions such as more efficient algorithms [16, 53, 59, 72], retiming using level-sensitive latches [34, 38, 46, 47, 60], and retiming for low power [31, 43].

Although no extra clock lines are necessary to borrow time like clock skew, retiming may change the position and number of flip-flops, making the design debugging process more difficult. Furthermore, time borrowing via retiming is inherently quantized because it is impossible to relocate a flip-flop to be in the middle of a logic gate if such granularity is necessary. The FPGA island-style architecture magnifies this potential problem because flip-flops can only move to fixed locations inside a CLB.

Another caveat of retiming is it ignores the initial state of sequential elements, which

(a)

(b)

(c)

Figure 2.7: Retiming example

is important for control circuitry. Fig. 2.8(a) shows a situation with two flip-flops that have different initial state values [52]. It appears that both can be pushed backwards to reduce flip-flop usage. However, Fig. 2.8(b) shows that if we do so, it is impossible to assign an initial state of “0” and “1” to the retimed flip-flop that satisfies the initial state requirements. Additional constraints are necessary to avoid such moves to ensure that retiming does not violate initial states. Touati and Brayton [70] present an algorithm that computes an equivalent initial state for the retimed circuit, if possible. Other work on ensuring an equivalent initial state include [40, 42, 65].

Figure 2.8: A retiming move that does not have an equivalent initial state

Retiming has been applied to FPGAs [12, 22, 63, 64, 74, 75]. Most recently, Singh [64] presented a linear-time algorithm that is aware of architectural, timing, legality, and user constraints. A 7% improvement in circuit speed was achieved.

Chapter 3

Level-Sensitive Latches

This chapter reviews the behaviour of a level-sensitive latch and its timing parameters in Section 3.1 and the timing constraints that ensure correct functionality of latch-based circuits in Section 3.2. We show how the constraints can be simplified in Section 3.3. We conclude by discussing prior work on using latches in Section 3.4.

3.1 Latch Basics

Level-sensitive latches are clocked sequential elements that are fundamental to building synchronous circuit designs. Fig. 3.1(a) shows the gate-level implementation and circuit symbol. The timing diagram in Fig. 3.1(b) shows a level-sensitive latch transfers data at the input (D) to the output (Q) whenever the clock is active. This is known as the *transparent* phase. Otherwise, the latch is *opaque* and Q holds its value. We will assume from here on in that a level-sensitive latch is transparent when the clock is high, and otherwise opaque.

The transparent nature of level-sensitive latches allow signals of one combinational stage to arrive before or during the transparent phase of the next clock cycle. This flexibility allows time borrowing to occur, thereby mimicking clock skew and retiming for flip-flop based circuits. The advantage of using level-sensitive latches is that they avoid

Figure 3.1: Latch basics

the dynamic power consumption overhead of using multiple clocks to implement clock skew and possible increase in the number of flip-flops if retiming is used.

These advantages come at a cost: timing analysis is more complex. The very ability that allows signals to arrive during the transparent phase of the subsequent clock cycle implies that the clock period is no longer determined by the longest path between sequential elements. Furthermore, not unlike a flip-flop, the latch itself has intrinsic delays and requires safety margins to ensure correct functionality.

Fig. 3.2 gives a pictorial comparison of the timing parameters for latches and flip-flops. The waveforms represent the clocks observed at each sequential element and arrows starting at one clock and terminating at another represent a combinational path. Striped regions adjacent to clock events, hold-time (T_{hold}) and setup-time (T_{su}), are timing windows that ensure the proper signal is captured, whereas solid regions, clock-to-Q delay (T_{cq}) and data-to-Q delay (T_{dq}), represent the intrinsic delays of a latch. The solid arrows show a possible long path that exploits the transparency of latches to borrow time, whereas the dashed purple arrows depicts a possible short path that starts at L1 and terminates at FF3. There are two notable differences between transparent latch and positive edge-triggered

Figure 3.2: Level-sensitive latch timing parameters

flip-flop timing parameters:

1. T_{su} and T_{hold} are bound to the falling edge of the clock rather than the rising edge.
2. T_{dq} only applies to latches during the transparent phase as flip-flops can only pass data from input to output on edge-triggered events.

Fig. 3.2 shows that level-sensitive latches allow signals to arrive at *any* time before the T_{su} timing window. This means a combinational stage only borrows what is necessary from a subsequent stage. Supporting varying amounts of time borrowing is analogous to clock skew’s need for multiple “skews” to satisfy different time borrowing requirements. In fact, level-sensitive latches driven by a *single* clock can mimic multiple “skews”.

The time borrowing properties of latches have definite advantages. However, minimum delays between sequential elements that may cause hold-time violations are still applicable to latch-based circuits. Fig. 3.3, first shown in the introduction, shows that the width of the transparent window is directly related to how susceptible a latch is to a hold-time

violation. We see that with the minimum and maximum combinational delays between sequential elements given in Fig. 3.3, it is possible that, a short path (dashed) can corrupt the data received at FF_3 . This is a hold-time violation. If we reduce the size of the transparent window for L_2 , it is possible to avoid this hold-time violation. To do this, the *pulse width*, which is the amount of time the clock is high during a cycle, must be altered. Latches that are driven by such clocks are referred to as *pulsed latches*.

Figure 3.3: The advantage of pulsed latches

Pulsed latches enable time borrowing like clock skew and retiming, while also providing a mechanism to avoid hold-time violations. The next section will talk about how the transparent nature of latches can be modeled using timing constraints.

3.2 Timing Constraints

Although Section 3.1 discusses the properties of a latch and their advantages, exploiting the transparency of latches requires satisfying timing constraints to ensure data still depart and arrive in relation to the clock. Table 3.1 summarizes the timing parameters of latches, which mostly resemble flip-flop timing parameters.

T_{cq}	clock-to-Q delay
T_{dq}	data-to-Q delay
a_i, A_i	earliest and latest arrival times at latch i
P	clock period
cd_{ji}, CD_{ji}	short and long $j \rightarrow i$ combinational path delay from latch j to latch i
T_{su}	setup time
T_h	hold-time
W_i	pulse width of latch i

Table 3.1: Summary of latch timing parameters

Latch timing constraints must ensure signals do not arrive *too* late. The combinational path represented by the solid arrows shown in Fig. 3.2 arrives at L_2 during its transparent phase. However, it must arrive before the T_{su} window of the falling edge of the clock. Equation (3.1) conveys this idea by modeling the latest arrival time, A_i , at latch i as a function of data arrival time at some latch j connected to i [33]:

$$A_i = \max_{\forall j \rightarrow i} [\max(T_{cq}, A_j + T_{dq}) + CD_{ji}], \forall i \quad (3.1)$$

A_i does not give any information on whether or not the latest signal has arrived too late. To ensure that a signal *never* arrives too late at a latch, we can bound it like so:

$$A_i \leq P + W_i - T_{su}, \forall i \quad (3.2)$$

That is, no signal can arrive later than T_{su} before the falling edge of the clock of the subsequent clock cycle. Combining (3.1) and (3.2), we obtain:

$$\max_{\forall j \rightarrow i} [\max(T_{cq}, A_j + T_{dq}) + CD_{ji}] \leq P + W_i - T_{su}, \forall i \quad (3.3)$$

The complex inequality shown in (3.3) ensures that *every* combinational path terminating at latch i must arrive before the T_{su} window bound to the falling edge of the clock

of the subsequent cycle.

As no sequential circuit is valid until considering hold-time constraints, we first describe the earliest arrival time of any signal at latch i , a_i :

$$a_i = \min_{\forall j \rightarrow i} [\max(T_{cq}, a_j + T_{dq}) + cd_{ji}], \forall i \quad (3.4)$$

Equation (3.4) describes a_i as a function of the arrival times at some latch i reachable by some $j \rightarrow i$ combinational path. Since a signal from latch j cannot launch *before* the T_{cq} window bound to the positive edge of the clock, the T_{cq} term provides a lower bound on data launch time from latch j . If data arrives at j during the transparent phase, an additional T_{dq} delay is necessary for data to be transferred from j 's input to output. After data leaves latch j , the minimum combinational delay necessary to arrive at latch i is modeled by cd_{ji} .

As the example given in Fig. 3.3 showed, data cannot arrive too early at latch i . Doing so would corrupt the intended data stored at other memory elements.

$$a_i \geq W_i + T_h, \forall i \quad (3.5)$$

Inequality (3.5) models a latch's hold-time constraint by enforcing all signals to arrive *after* latch i 's transparent window closes in the current cycle. Combining (3.4) and (3.5) yields:

$$\min_{\forall j \rightarrow i} [\max(T_{cq}, a_j + T_{dq}) + cd_{ji}] \geq W_i + T_h, \forall i \quad (3.6)$$

The combined hold-time constraint given in (3.6) ensures that no short path launching from any latch connected i arrives during i 's window of transparency, W_i . As T_{cq} , T_{dq} , and T_h are latch timing parameters, the only variables are cd_{ji} and W_i . We initially tackle timing optimization on an already routed FPGA design. Therefore, W_i is the only variable we have control over and it must be carefully selected so that no hold-time violations

arise, while also providing the maximal time borrowing benefits if only one clock is used.

But before selection of the pulse width can occur, we need to calculate the clock period of latch-based circuits. The *max* and *min* terms in (3.3) and (3.6) respectively prevent the use of conventional optimization approaches, such as linear programming and graph algorithms. We simplify the constraints to allow the use of conventional optimization techniques in the next section.

3.3 Simplifying Latch Timing Constraints

For us to apply conventional optimization approaches to find the clock period of latch-based circuits, the latch timing constraints discussed in the previous section must be simplified first. Starting with (3.3), we can remove the leftmost *max* term by constructing a constraint for every $j \rightarrow i$ path, rather than using one constraint to represent *all* paths terminating at latch i :

$$\max(T_{cq}, A_j + T_{dq}) + CD_{ji} \leq P + W_i - T_{su}, \forall j \rightarrow i \quad (3.7)$$

The purpose of the remaining *max* term is to ensure that the signal at latch j launches no earlier than T_{cq} after the rising edge. We can represent (3.7) with two constraints:

$$A_j + T_{dq} + CD_{ji} \leq P + W_i - T_{su}, \forall j \rightarrow i \quad (3.8)$$

$$A_j + T_{dq} \geq T_{cq}, \forall j \quad (3.9)$$

Inequality (3.9) is a lower bound on the launch time of a signal from latch j . (3.8) and (3.9), although simplified, still contain 3 variables: A_j , P , and W_i . We can remove A_j by conservatively assuming that the latest arrival time at latch j always occurs at the falling

edge of a pulse, that is $A_j = W_j - T_{su}$. Plugging this into (3.8) and (3.9) gives:

$$W_j + T_{dq} + CD_{ji} \leq P + W_i, \forall j \rightarrow i \quad (3.10)$$

$$W_j - T_{su} + T_{dq} \geq T_{cq}, \forall j \quad (3.11)$$

Similarly, the hold-time constraint for latches given in (3.6) can be relaxed by first transforming (3.6) to occur between every latch pair connected by a combinational path, just like the relaxation process used for (3.7):

$$\max(T_{cq}, a_j + T_{dq}) + cd_{ji} \geq W_i + T_h, \forall j \rightarrow i \quad (3.12)$$

We can conservatively assume that every early signal launches at the beginning of a latch's opening window (i.e. the rising edge of the clock). Based on this assumption, we set $a_j = 0$, resulting in:

$$\max(T_{cq}, T_{dq}) + cd_{ji} \geq W_i + T_h, \forall j \rightarrow i \quad (3.13)$$

As T_{cq} and T_{dq} are fixed for a specific latch design, they are fixed during the optimization process. Therefore, we can replace the $\max$ term with the larger of the two timing parameters (assuming $T_{cq} \geq T_{dq}$ in this case):

$$T_{cq} + cd_{ji} \geq W_i + T_h, \forall j \rightarrow i \quad (3.14)$$

Although simplifying (3.10) and (3.14) would appear to restrict the full potential of using latches, we will show that one clock can still achieve measurable gains under these assumptions.

3.4 Prior Work

Timing optimization of latch-based circuits has been studied extensively for ASICs. Most prior work has formulated the latch-based optimization problem using linear constraints and solved it using linear programming (LP) [38, 50, 51, 66, 67] or graph algorithms [23, 58]. Among the prior work using transparent latches, our approach is most similar to [23]. The authors optimize circuit performance by using two clocks with adjustable duty cycles. Their approach is exact and can be extended to more than two clocks. However, they strictly forbid combinational paths that start and end at the same latch, which we found to be quite prevalent in our benchmark suite. Our formulation supports these combinational paths, while also improving performance using only a single clock.

Pulsed latches are widely used in microprocessors for better performance [3, 9, 28, 30, 44, 54]. Their use for improving the performance of ASICs in general has also been explored recently by Lee et. al [33, 34, 46]. Using flip-flop-like timing constraints, their optimization strategy relies on exploiting the *difference* between pulse widths and clock delays to steal time from neighboring combinational stages. Their approach to time borrowing uses multiple pulse widths and skewed clocks. This differs from our approach which mimics the presence of multiple “skews” using *one* pulse width.

Chapter 4

Graph-Theoretic Timing Optimization

Linearizing latch timing constraints, as discussed in Section 3.3, allows us to solve for the optimal clock period of a circuit using well-studied analytic methods such as linear programming, or graph-based approaches. Section 4.1 reviews standard graph terminology. Section 4.2 discusses the background and intuition on what the optimal clock period represents in our graph formulation. We introduce how long path, short path, and special constraints are handled in our formulation. Finally, Section 4.3 introduces Howard’s algorithm. We use this algorithm to calculate the optimal clock period of a circuit.

4.1 Preliminaries

Before the graph formulation is described, some basic graph terminology must first be defined. Let $G = (V, E)$ be a strongly-connected directed graph. Let a *vertex* $v \in V$ represent a flip-flop or a latch in G . Every v has an associated W_v , the pulse width. Let an *edge*, $e(u, v)$, and its delay, $d(u, v)$ represent the maximum delay on a $u \rightarrow v$ combinational path. A *path* is a traversal of vertices through connecting edges with an arbitrary start and end vertex. A *cycle* is a path that starts and ends at the same vertex.

Let c and C represent a cycle and the set of *all* cycles in G , respectively.

4.2 Calculating The Optimal Clock Period

Linearizing the latch timing constraints enables the use of linear programming or existing graph algorithms for optimizing the clock period. We show how the clock period can be analytically calculated when considering only long path constraints in Section 4.2.1. We then extend the formulation to handle short path and other special constraints in Section 4.2.2, and show how these constraints can model a critical I/O path in Section 4.2.3.

4.2.1 Long Path Constraints

We show in this section how to map the latch long path (setup-time) constraint, restated below, into our graph-theoretic model:

$$W_j + T_{dq} + CD_{ji} - P \leq W_i, \forall j \rightarrow i \quad (4.1)$$

We create two vertices v_j and v_i , an edge $e(j, i)$ with $d(j, i) = T_{dq} + CD_{ji} - P$ to represent the relationship between latches j and i . The optimization objective is to find the minimum P such that a function that maps a valid value W_i to each v_i exists. We refer to this function as ω . Such a formulation is known as the *parametric shortest path* problem [24, 80].

We note that if P were a constant, the problem reduces down to a standard shortest path problem and Bellman-Ford can be applied to find a feasible ω . One common approach [33, 34, 62–64] is to test different values of P using binary search and solve the system using Bellman-Ford. Binary search can be used because Bellman-Ford would not be able to return a feasible ω if P were too low.

Rather than using binary search along with Bellman-Ford to find the optimal clock

period, P_{opt} , we solve for it analytically. Karp and Orlin [24] observed that a solution to the parametric shortest path problem can be found by calculating the *maximum mean cycle* (MMC) of G . To do so, we set $d(j, i) = T_{dq} + CD_{ji}$ and “encode” P directly onto the edge. That is, the mere existence of an edge signifies that a constraint is a function of P . More formally, we define the MMC to be:

$$MMC(G) = \max_{c \in C} \left[\sum_{e(u,v) \in c} \frac{d(u,v)}{|c|} \right] \quad (4.2)$$

where $|c|$ represents the number of edges on c . In essence, the MMC is the maximum total delay of any cycle in G divided by the number of sequential elements on that cycle. It represents the best clock period that can be achieved by latches/retiming/clock skew for the circuit, given the specified delay values. A proof of this property can be found in [2].

Figure 4.1: Sample circuit fragment and its graph representation

Fig. 4.1 shows how a circuit fragment shown in Fig. 4.1(a) is represented in our graph formulation depicted in Fig. 4.1(b). Edges are labeled with the circuit’s longest combinational path delays between sequential elements. Fig. 4.1(b) also illustrates how the $MCR(G)$ yields the P_{opt} . For this example, assume that every edge in Fig. 4.1(b) represents constraint (4.1). The cycle containing dashed edges, $v_A \rightarrow v_D \rightarrow v_C \rightarrow v_A$,

is the MMC in this example, with a value of 5: the sum of edge delays along the cycle is 15, and $15/(3 \text{ edges}) = 5$. Thus, we can operate this circuit with a clock period of 5. Observe, however, that the edge from $v_A \rightarrow v_D$ has a path delay of 7. Consequently, we must set $W_D = +2$, in order for signals on the $v_A \rightarrow v_D$ path to arrive on time. In addition, since timing analysis requires *EVERY* constraint to be satisfied, we must set $W_B = +1$ to satisfy $v_A \rightarrow v_B$. Any other cycle in this graph, with a lower cycle ratio would not satisfy every constraint.

4.2.2 Short Path and Special Constraints

The problem with the current MMC definition is that every edge in G must be parameterized. The latch short path constraint (3.14) does not depend on P . Such a constraint would lead to a *non-parameterized* edge. To handle these, our graph formulation must be extended. The initial proposition that P_{opt} can be calculated by finding $\text{MMC}(G)$, with P encoded onto the edge is no longer valid because some edges are not parameterized to begin with. To account for non-parameterized edges, we introduce the idea of *weight*, $w(u, v)$, of an edge $e(u, v)$. Parameterized edges (long paths) have $w(u, v) = 1$, whereas non-parameterized ones (short paths) are represented by $w(u, v) = 0$. That is, we encode whether or not an edge is parameterized into $w(u, v)$. By calculating the *maximum cycle ratio* (MCR) of G considers not only long path constraints, but short paths as well. Therefore, finding the $\text{MCR}(G)$ is equivalent to finding the optimal clock period of a circuit while considering short paths. More formally, we define the MCR to be:

$$\text{MCR}(G) = \max_{c \in C} \left[\frac{\sum_{e(u,v) \in c} d(u, v)}{\sum_{e(u,v) \in c} w(u, v)} \right] \quad (4.3)$$

By construction, cycles with zero weight and non-zero delay in G do not exist. This avoids the situation where $\text{MCR}(G) = \infty$ when in fact, a valid cycle exists. Furthermore, this generalized model allows the secondary long path constraint (3.11) to be modeled,

and it gives us the ability to artificially restrict W for certain nodes such as inputs and outputs. We discuss this topic in the next section.

4.2.3 Input-Output Paths

Section 4.2.2 showed that finding the MCR of a circuit's graph representation will yield the circuit's optimal clock period. However, G must be strongly-connected. Why is this important? Consider G given in Fig. 4.2(a), with sequential elements represented by v_A, v_B, v_C, v_D , input pin v_I , and output pin v_O . This graph is *not* strongly-connected because no vertex can reach v_I via some path. Using this graph, we observe that $\text{MCR}(G) = 5$, due to the cycle highlighted by dashed edges, and conclude that $P_{opt} = 5$. However, this cannot satisfy the combinational path $v_I \rightarrow v_A \rightarrow v_B \rightarrow v_O$, because its total delay is 18 time units, and extends across 3 combinational stages. Therefore, each stage on average requires 6 time units to complete, and $P_{opt} = 5$ would be insufficient.

Modifying G to be strongly-connected can rectify this problem, as shown in Fig. 4.2(b). This can be done by adding an artificial node we call the *supernode*, v_S , and a pair of edges connecting v_S and every I/O vertex. To ensure I/O vertices do not borrow time, we construct a pair of edges between v_S and v_I , $e(i, s)$ and $e(s, i)$, as well as between v_S and v_O , $e(o, s)$ and $e(s, o)$, as shown in Fig. 4.2(b). We set $d(u, v) = 0$ and $w(u, v) = 0$ to represent equality. With these additional constraints, $\text{MCR}(G)$ is now 6 because a cycle including the input and output vertex can be formed now through v_S .

4.3 Howard's Algorithm

The equations and constraints defined in Section 3.3 can be solved using linear programming. However, we use Howard's algorithm [10] to find the MCR, as it is believed to be the fastest MCR algorithm in practice [13] with a near-linear runtime with respect to the number of edges in G .

Figure 4.2: Detecting critical I/O paths

Howard's algorithm takes an iterative approach to computing the MCR. The algorithm always operates on the *policy graph* of G , G_p . G_p is simply a subgraph of G and it provides a fast way to find candidate cycle ratios, r , that may or may not be the actual MCR. To verify this, the arrival times, A_i , at each node are calculated for the given r . This is analogous to finding the pulse widths or node latencies for a circuit after the MCR has been found. Finding candidate cycles in G_p and computing A_i together are known as *value determination*. Using the computed arrival times and cycle ratio(s), *policy improvement* is employed to determine whether or not r is indeed the MCR. If r is not the MCR, policy improvement mutates G_p in such a way that the subsequent r values monotonically approach the final MCR.

Howard's algorithm is best interpreted with the aid of an example. We will explain the two major steps of Howard's while walking the example given in Fig. 4.3.

4.3.1 Value Determination

Value determination is responsible for finding candidate cycles, computing their cycle ratios, and propagating arrival times. First, G_p must be constructed. Howard's algorithm requires that G_p contains every vertex of G , but each vertex must have *exactly* one incoming edge. The solid edges shown in Fig. 4.3(b) gives a possible configuration of G_p .

Figure 4.3: Howard’s Algorithm

To compute a candidate r value, a cycle c must be found in G_p ¹. This can be achieved by visiting vertices in backwards topological order starting from some arbitrary vertex. If a cycle cannot be found, the process starts again from another vertex. Once c is found, r is computed by the following equation:

$$r = \frac{\sum_{e(u,v) \in c} d(u, v)}{\sum_{e(u,v) \in c} w(u, v)} \quad (4.4)$$

Equation (4.4) simply states that r is the sum of the delays divided by the sum of the weights around c . Now that we know how to calculate r , r needs to be tested to see if it actually is the $\text{MCR}(G)$. Testing r requires computing arrival times, A_i , at every node. Let's assume that each edge has weight $w(u, v) = 1$ in this example. For instance, the self-terminating loop at v_B has $r = 3$. Given r , A_i for all reachable nodes from v_B must be computed, in a breadth-first search-like manner. Furthermore, each vertex is labeled by the r value of the cycle that it is reachable from. This is necessary for policy improvement. We first set $A_B = 0$ because it is where we begin the propagation. A_i for some reachable v_i can be calculated using:

$$A_i = d(j, i) + A_j - w(j, i) \times r \quad (4.5)$$

To build some intuition on what equation (4.5) represents, if $w(j, i) = 1$, we can view r as the clock period of a conventional flip-flop-based circuit. A_i would represent the arrival time with respect to positive edge of the clock. That is, a negative A_i value indicates that a signal will arrive before the positive edge, whereas a positive value indicates arriving after. Fig. 4.3(b) shows that v_C and v_D can be reached from v_B . Their arrival times are 1 and -1, respectively. What about v_A ? It clearly cannot be reached from v_B via the current edges in G_p (solid edges). Value determination deals with this by selecting a

¹If G is not strongly connected, G_p may not contain a cycle. If so, the algorithm terminates without a solution. However, we ensure G is strongly-connected before calling Howard's algorithm.

new cycle, calculating r , and labeling reachable vertices for all remaining unreachable vertices, until all are reachable from some cycle. The table shown in Fig. 4.3(b) gives the computed r and A_i values for every node.

4.3.2 Policy Improvement

After value determination has labeled every vertex, policy improvement must determine if the proposed cycle ratio is the MCR. The cycle ratios proposed in the table of Fig. 4.3(b) cannot be the MCR simply because not every vertex agrees on the same cycle ratio. There can only be one *maximum* cycle ratio. The algorithm rectifies this problem by mutating G_p . Specifically, the incoming edge of v_A must be replaced as v_A 's cycle ratio is too low. The two dashed edges shown in Fig. 4.3(b) are replacement candidates. The standard algorithm does not bias towards any specific candidate. Suppose $e(B, A)$ is selected for the next iteration.

Fig. 4.3(c) shows another iteration of Howard's algorithm. After arrival times and cycle ratios have been computed, every vertex now agrees that $r = 3$. Policy improvement then verifies that A_i for every node in G_p satisfies *every* edge in G by computing a new set of arrival times, L_i , using the following equation:

$$L_i = \max_{j \rightarrow i} [d(j, i) + A_j - w(j, i) \times r], \forall i \quad (4.6)$$

L_i is the latest possible arrival time at v_i , when considering all edges in G . Finding an edge that can generate a later arrival time than the calculated A_i value implies that the current r value is derived from incomplete information, and therefore cannot be the MCR. G_p is mutated to replace existing incoming edges with the new edges that generated later arrival times, according to (4.6). The dashed edges shown in Fig. 4.3(c) are examples of such edges.

After skipping an iteration of the algorithm, we arrive at the version of G_p depicted

in Fig. 4.3(d). The cycle $v_B \rightarrow v_C \rightarrow v_B$ has $r = 5.5$. Every node is reachable from c . Furthermore, L_i does not differ from A_i for any node. Policy improvement concludes at this point that $r = \text{MCR}(G)$.

Chapter 5

Latch-Based Timing Optimization

We discuss in Section 5.1 our initial work [68] that replaces certain flip-flops with latches for better timing performance. In Section 5.2, we discuss an additional optimization that can increase the pulse width through better avoidance of short paths, leading to even better performance for certain circuits. Section 5.3 discusses situations where maximizing the pulse width would not always give the best results. Section 5.4 presents the results of our latch-based optimizations.

5.1 Post Place and Route Latch Insertion

Although we can incorporate the hold-time constraint shown in (3.14) directly into G and calculate P by finding $\text{MCR}(G)$, the solution to this problem would imply that every sequential element is a latch with a (possibly different) pulse width, thereby requiring multiple clocks and significant power consumption. We wish to use a single clock and therefore, must decide on a specific pulse width to use, and also whether each sequential element should be a flip-flop or a latch. Although flip-flops cannot borrow time, their hold-time constraints are less restrictive and because of this, we use them to prevent very short paths from limiting the pulse width. The flip-flop/latch choice adds a binary decision element to the optimization problem. We explore the use of a greedy heuristic

that maximizes the pulse width.

We first solve for the best-case clock period, P_{init} , and associated pulse widths, ω_{init} , without considering hold-time constraint (3.14), and then use ω_{init} in conjunction with (3.14) to guide the process of selecting some sequential elements to be latches, some to remain as flip-flops and settle on a *single* pulse width, W_{final} . The approach we take is to start with a large value for W_{final} and then scale it back based on any short path delay violations. We then assign each sequential element to be either a flip-flop or a latch, and then re-solve for P and W_{final} based on the flip-flop/latch assignments. The full algorithm is detailed in Algorithm 1.

Algorithm 1 Pulsed latch timing optimization

Input: $G(V, E)$, E_{min}

Output: $P_{final}, W_{final}, \omega_{final}$

- 1: $P_{init}, \omega_{init} \leftarrow \text{Howard}(G)$
 - 2: Sort edges in E in ascending order of their short path delays in E_{min}
 - 3: $W_{final} \leftarrow \text{FindPulseWidth}(\omega_{init}, G, E_{min})$
 - 4: $\text{SetSeqElements}(W_{final}, G, E_{min})$
 - 5: $P_{final}, \omega_{final} \leftarrow \text{Howard}(G)$
-

Algorithm 2 FindPulseWidth

Input: ω, E_{min}

Output: W_{final}

- 1: $W_{final} \leftarrow \max(\omega)$
 - 2: **for** $e(u, v) \in \text{sorted } E$ **do**
 - 3: $d_{min}(u, v) \leftarrow \text{short path delay of } e(u, v) \text{ from } E_{min}$
 - 4: **if** $T_{cq} + d_{min}(u, v) < \omega(v) + T_h$ **then**
 - 5: $W_{final} \leftarrow T_{cq} + d_{min}(u, v) - T_h$
 - 6: **break**
 - 7: **end if**
 - 8: **end for**
-

The inputs to Algorithm 1 are $G(V, E)$, which represents the constraint set depicted in Fig. 4.1, and the set of minimum delays between every pair of sequential elements, E_{min} . At line (1), we begin by calculating P_{init} and ω_{init} subject to only the constraints described in Section 4.2.1 (no consideration of short path delays). At line (2), we sort

Algorithm 3 SetSequentialElements**Input:** W_{final} , G , E_{min}

```

1: for  $e(u, v) \in E$  do
2:    $d_{min}(u, v) \leftarrow$  short path delay of  $e(u, v)$  from  $E_{min}$ 
3:   if  $T_{cq} + d_{min}(u, v) < W_{final} + T_h$  then
4:     forceFlipFlop( $v$ )
5:   else
6:     forceLatch( $v$ )
7:   end if
8: end for

```

the edges in E in ascending order of the values in E_{min} . This step identifies the first possible hold-time violation without having to iterate through all of E_{min} . Line (3) calls FindPulseWidth, Algorithm 2, to heuristically calculate W_{final} subject to short path constraints. Algorithm 2 iterates in ascending order through the short path constraints imposed, until we encounter the first violation of constraint (3.14). If we do find a violation, we know that using a pulse width larger than $T_{cq} + d(u, v) - T_h$ would not satisfy the violated constraint. Furthermore, subsequent edges in the sorted edge set would not generate a more constraining pulse width because the edge set is sorted in ascending order. We set W_{final} at line (5) and return to Algorithm 1. In Algorithm 1, the computed W_{final} value is used by SetSeqElements at line (4). Algorithm 3 constrains sequential elements that have incoming edges with minimum delay less than W_{final} to be flip-flops to prevent hold-time violations from occurring. Otherwise, we allow them to be latches so that timing borrowing is possible, if required. Taking into account W_{final} and sequential element type, P_{final} and ω_{final} are computed at line (5).

5.2 Iterative Improvement

The algorithm discussed in Section 5.1 selects W_{final} based on the first latch hold-time violation it encounters due to some short path. Re-solving for P_{final} yields a set of latencies and a clock period that obeys all constraints.

The problem with simply one iteration is that the decision on W_{final} depends on the relationship between a short path and the flip-flop/latch such a path terminates at. The latencies, ω_{init} , that these short paths encounter is a set necessary to implement the *optimal* clock period, in the absence of hold-time constraints. Implementing a lower clock period requires more time borrowing across the circuit, but also makes the short path constraint harder to satisfy. The purpose of the iterative approach is to start with a conservative estimate of P_{final} and ω_{final} , which is given by the algorithm described in Section 5.1, and try to reduce P by gradually increasing W . How can we do this? By re-examining the assumptions on which sequential elements had to remain as a latch, we may find some latches that had to borrow time only when trying to implement P_{init} , but do not with a more conservative P and ω . By converting such latches to flip-flops we can satisfy more hold-time constraints, leading to the possibility that a larger W can be used for other sequential elements, enabling more time borrowing.

Algorithm 4 Iterative improvement using pulsed latches

Input: $G(V, E)$, E_{min}
Output: P_{incr} , W_{incr} , ω_{incr}

```

1:  $P_{prev} \leftarrow \infty$ 
2:  $P_{init}, \omega_{init} \leftarrow \text{Howard}(G)$ 
3: Sort edges in  $E$  in ascending order of their short path delays in  $E_{min}$ 
4:  $W_{incr} \leftarrow \text{FindPulseWidth}(\omega_{init}, G, E_{min})$ 
5: loop
6:    $\text{SetSeqElements}(W_{incr}, G, E_{min})$ 
7:    $P_{incr}, \omega_{incr} \leftarrow \text{Howard}(G)$ 
8:   if  $P_{incr} \geq P_{prev}$  then
9:     break
10:  end if
11:   $W_{incr} \leftarrow \text{FindNewPulseWidth}(G, E_{min}, W_{incr})$ 
12:   $P_{prev} \leftarrow P_{incr}$ 
13: end loop

```

Aside from initializing P_{prev} , lines (1) - (7) in Algorithm 4 are identical to Algorithm 1. The check at lines (7) - (9) ensures that the newly calculated clock period did not get worse when compared to the previous iteration's clock period. Line (11) selects a new

pulse width for the circuit. Our pulse width selection strategy is to use the next shortest short path to generate a new candidate pulse width, W_{incr} , as shown in Algorithm 5. This method, although not as aggressive as the heuristic approach, can lead to better solutions. An example of such a scenario is described in Section 5.3.

P_{prev} is updated at line (12) in preparation for the next iteration. Using the new W_{incr} , the appropriate sequential elements are converted to flip-flops to avoid hold-time violations and subsequently solved using Howard’s algorithm. Iterating through these steps continues until no more improvements are possible.

Algorithm 5 FindNewPulseWidth

Input: $G(V, E)$, E_{min} , W_{incr}
Output: W_{incr}

```

for  $e(u, v) \in$  sorted  $E$  do
     $d_{min}(u, v) \leftarrow$  short path delay of  $e(u, v)$  from  $E_{min}$ 
    if  $T_{cq} + d_{min}(u, v) - T_h > W_{incr}$  then
         $W_{incr} \leftarrow T_{cq} + d_{min}(u, v) - T_h$ 
        break
    end if
end for

```

Fig. 5.1 gives an example that shows how the original pulsed latch timing optimization discussed in Section 5.1 can improve the performance of conventional flip-flop designs. It also shows how the performance can be further improved with the iterative approach discussed in this section. Fig. 5.1(a) shows a simple circuit represented by I/O, v_I and v_O , and 4 sequential elements, v_A , v_B , v_C , and v_D . Ignoring short path constraints, the circuit can operate at $P_{opt} = 5$, with the I/O path $v_I \rightarrow v_B \rightarrow v_C \rightarrow v_O$ limiting performance. In contrast, the circuit could only operate with a clock period of 9 time units without time borrowing due to $v_I \rightarrow v_B$. Suppose that the shortest short path shown in Fig. 5.1(b) as the dotted arrow terminating at v_A has a minimum delay of 1. To implement the optimal clock period, the table in Fig. 5.1(a) shows that v_A needs to borrow 2 time units. This short path would cause a hold-time violation at v_A . To prevent this hold-time violation from occurring, v_A is limited to borrow no more than 1 time unit. Therefore, we must

Figure 5.1: Iterative improvement example

settle on a pulse width $W = 1$ for the circuit. With this time borrowing constraint, a clock period of 8 is achievable, as shown in Fig. 5.1(b). This summarizes the optimization discussed in Section 5.1.

Upon further inspection, $W = 1$ was only necessary because v_A had to borrow time to implement P_{opt} . After re-calculating the latencies, we see that v_A does not require time borrowing at all with $P = 8$. Therefore, the short path terminating at v_A should not cause a hold-time violation if we force v_A to be a flip-flop. So if the short path that caused us to settle on $W = 1$ can be avoided by forcing v_A to be a flip-flop, we use the next shortest short path, which terminates at v_D as shown in Fig. 5.1(c), to generate a new candidate pulse width. In this case, we set $W = 2$. Using this assumption, the appropriate sequential elements are forced to be flip-flops and by re-solving for P , we can verify if a lower clock period is feasible. Fig. 5.1(d) shows that a clock period of 7 is achievable, which is lower than initial clock period of 8. This process continues until no improvements to the clock period can be made.

5.3 Optimality

Why do we choose such a safe method for selecting a new W in our iterative approach? Although the heuristic pulse width selection approach shown in Algorithm 1 provides a fast way to settle on a good pulse width, it may or not lead to the best achievable clock period with a single pulse width. In contrast, the iterative approach slowly converges on the best clock period by trying all pulse widths subject to minimum delays. To show that the heuristic approach can miss solutions found by the iterative approach, we will walk through an example that shows maximizing the pulse width will not always give the optimal clock period achievable with a single pulse width. Consider the sample circuit fragment given in Fig. 5.2 with long path (solid edges) and short path (dashed edges) constraints. Suppose that the input (v_I) to output (v_O) path yields the MCR.

(For conciseness, v_S is not shown). A clock period of $\frac{7+4+7+2}{4} = 5$ time units is sufficient and results in the given time borrowing requirements, W_A, W_B , and W_C . Based on the heuristic given in Section 5.1, a suitable pulse width must be chosen subject to the minimum delay constraints (dashed edges) given in Fig. 5.2(b). Thus, constraint (3.14) purely depends on the relationship between cd_{ji} and W_i . Recall that the heuristic given in Algorithm 1 processes minimum delays in ascending order. Therefore, the first path that gets processed is the one incident on v_A . Since this path does not cause a short path violation, the search for W_{final} continues. Once the heuristic processes the path incident on v_C , this sets W_{final} to 2.5 units. With $W_{final} = 2.5$, v_A and v_B are forced to be flip-flops. This results in a clock period of 7 time units (due to $v_I \rightarrow v_A$), as shown in Fig. 5.2(c).

Figure 5.2: Contrasting the greedy and iterative pulse width selection approaches

In contrast, the iterative approach would settle on $W_{final} = 2.25$, leading to a clock period of 5.25, as shown in Fig. 5.2(d).

5.4 Experimental Study

We implemented our approach within the VPR 5.0 framework [39]. Our results use a mix of VPR 5.0 and MCNC benchmarks mapped to an architecture using a LUT size of 6, and cluster size of 8. We used an island style FPGA architecture consisting of 50% length-4 and 50% length-2 routing segments. Each benchmark was given an extra 30% in channel capacity on top of the minimum required for routing to emulate a “medium stress” scenario. The results for each circuit are averaged over four runs, each using a different placement seed value. We set $T_h = \frac{1}{2}T_{cq}$, and $T_{cq} = T_{dq} = T_{su}$, based on values for the Xilinx Virtex 6.

In this work, we show the impact of latches and clock skew both with and without considering hold-time violations. Naturally, smaller short path delays lead to more hold-time violations and degraded performance results. However, the VPR timing model does not incorporate short path delays for combinational logic and routing paths. To handle this, we present results for several short path delay scenarios, ranging from optimistic to pessimistic. Specifically, we use VPR’s timing information to calculate the shortest paths between sequential elements and emulate different scenarios by taking a fraction, $f\%$, of these short delays. We consider three settings for f : 80% (optimistic), 70% (medium), and 60% (pessimistic).

Table 5.1 contrasts the gains of using pulsed latches and flip-flops with the heuristic described in Section 5.1 (columns labeled PL_{heur}), pulsed latches with iterative improvement (columns labeled PL_{iter}), clock skew using flip-flops only (columns labeled CS) with no restrictions on the number of clock lines available, and theoretical possible gains using latches (column labeled PL_{opt}) without *any* short path constraints. The column labeled “Critical Path” presents the clock period of each circuit assuming only flip-flops are used, and the flip-flops are driven by a single clock. We refer to this as the “traditional” flow. The results in the table, in essence, represent different ways of analyzing a set of fully placed and routed designs. Values in the table represent achievable clock periods in *ns*,

as reported by VPR and our latch-based timing analysis framework. Geometric mean results and normalized geometric means appear at the bottom of each column.

On average, the “ PL_{opt} ” column shows that clock periods can be reduced by 32%. Since this result does not consider hold-time violations, it represents a lower bound on the achievable clock period. The rest of the results in Table 5.1 consider hold-time violations. For example, the columns grouped under 80% represent results for the case of minimum path delays set to be 80% of VPR’s minimum path delays. Observe that the performance results degrade considerably when hold-times constraints must be honored, and underscores the necessity of considering such constraints. The heuristic pulsed latch timing optimization provides a 5% performance improvement, on average, relative to the traditional flow. Clock skew with arbitrarily many clocks and pulsed latches with iterative improvement both provide a 9% improvement. Similar results are observed when minimum delays are set to 70% and 60%, although the gains relative to the “traditional flow” start to diminish.

Although not shown for each benchmarks, the run-time of PL_{heur} was in the order of a few seconds. However, iterative improvement can take upwards of one hour if it has any benefits due to repeated calls to Howard’s algorithm.

For certain benchmarks, the optimal clock period was achieved by pulsed latches under all minimum delay assumptions. This is usually caused by two inherent circuit properties. First, if all long path combinational delays between sequential elements are very balanced, then not much time borrowing is necessary to balance delays between combinational stages. Therefore, short paths are unlikely to restrict the amount of time borrowing necessary to achieve the optimal clock period. A second cause is that our formulation assumes that I/O cannot borrow time, as discussed in Section 4.2.3. This was ensured by constructing the supernode and a pair of edges between the supernode and every I/O node. It is possible that a critical long path that determines the clock period terminates at an output node. Because an output node cannot borrow time, the

presence of short paths cannot limit gains.

The more surprising result is that iterative improvement using pulsed latches with a *single* clock can match clock skew's theoretical gains for most benchmarks. To understand how this is possible, we recall that inequalities (5.1) and (5.2), restated below, model the pulsed latch and clock skew hold-time constraints, respectively.

$$T_{cq} + cd_{ji} \geq W_i + T_h, \forall j \rightarrow i \quad (5.1)$$

$$D_j + T_{cq} + cd_{ji} \geq D_i + T_h, \forall j \rightarrow i \quad (5.2)$$

The difference here is that clock skew's hold-time constraint is easier to satisfy due the ability to increase the D_j term in (5.2). This corresponds to shifting the clock driving a short path's starting flip-flop forward. Fig. 5.3(a) shows a circuit fragment with three flip-flops that illustrates this advantage. Suppose that this fragment limits the performance of the whole circuit. Without any time borrowing, the performance is limited by the 8 time unit long path between FF_1 and FF_2 . However, clock skew can reduce the clock period down to 5 time units if FF_2 and FF_3 's clock signals can be delayed by 3 and 2 time units, respectively. Furthermore, the dashed arrows in the timing diagram show that short path constraints are obeyed. If we attempt to reduce the clock period with pulsed latches using a single pulse width, a clock period lower than 6 cannot be achieved. Doing so would require FF_2 to borrow more than 2 units of time. This forces FF_3 to be a latch, and thus would be suboptimal because the short path terminating at FF_3 will limit time borrowing to 1 time unit for the whole circuit. Using clock skew to borrow time avoids the constraints of this short path because FF_2 got shifted forward, which delays FF_2 's short path launch time.

Fig. 5.3(b) shows how a small change to the short path delay between FF_1 and FF_2 can nullify the advantage clock skew has over pulsed latches. The reduction essentially

requires that the *difference* in skews between FF_1 and FF_2 be no larger than 1 time unit. Otherwise, a hold-time violation would occur. With this modification, clock skew can only achieve a clock period of 7 time units, which is no better than pulsed latches.

Figure 5.3: Illustrating the advantage of clock skew over pulse latches and its limitations

Despite the advantage illustrated by Fig. 5.3(a), our results in Table 5.1 indicate very few benchmarks where clock skew can do better than pulsed latches. To understand why, we must look beyond the simple example shown in Fig. 5.3(a) that demonstrates clock skew's ability to better avoid short paths. This advantage starts to diminish when we consider that multiple paths may fan in and fan out from a sequential element. If a short path with a delay of 1 terminates at FF_2 that starts from some other flip-flop k , k must borrow *at least* 2 time units just so that FF_2 can borrow 3 time units. The problem with this is that the long paths starting from k have *less* to complete simply because FF_2 needs to borrow time. This may be undesirable if the later launch time of k 's long paths actually puts one of them on the critical path. This effect only gets worse as the fan in and/or fan out of FF_2 and adjacent flip-flops increase.

Minimum Delay Assumptions		None		80%		70%		60%		
Clock Period (ns)	Critical Path	PL_{Opt}	PL_{heur}	PL_{iter}	CS	PL_{heur}	PL_{iter}	PL_{heur}	PL_{iter}	CS
paj_bounded_hierarchy	6.002	5.348	5.659	5.659	5.659	5.714	5.714	5.77	5.77	5.77
sv_chip0_hierarchy	4.07	3.389	3.726	3.726	3.724	3.782	3.782	3.837	3.837	3.835
clma	7.3	6.942	7.065	7.065	7.142	7.079	7.079	7.093	7.093	7.142
tseng	4.628	3.242	4.285	4.285	4.285	4.34	4.34	4.396	4.396	4.396
fir_scu_rtl_restructured	14.836	5.194	14.492	14.492	14.492	14.548	14.548	14.603	14.603	14.603
des_perf	3.305	2.316	2.878	2.878	2.966	2.933	2.933	2.989	2.989	3.07
iir	15.939	6.164	15.596	15.017	14.905	15.651	15.121	15.707	15.226	15.173
sv_chip1_hierarchy	12.06	3.958	11.717	8.674	8.574	11.772	8.674	11.828	8.674	8.636
des_area	5.638	5.422	5.422	5.422	5.622	5.422	5.422	5.422	5.422	5.622
iir1	15.524	14.158	15.181	15.181	15.181	15.237	15.237	15.292	15.292	15.292
cf_cordic_v.8.8.8	3.031	1.783	2.687	2.687	2.687	2.743	2.743	2.798	2.798	2.798
paj_raygentop_hierarchy	11.394	5.797	11.05	8.357	8.153	11.106	8.357	11.161	8.357	8.241
macl	12.306	11.567	11.962	11.962	11.962	12.018	12.018	12.073	12.073	12.073
elliptic	4.481	3.472	4.137	4.137	4.137	4.193	4.193	4.248	4.248	4.248
rs_decoder_1	14.078	13.045	13.734	13.734	13.734	13.79	13.79	13.845	13.845	13.845
cf_cordic_v.18.18.18	5.197	3.021	4.853	4.853	4.853	4.909	4.909	4.964	4.964	4.964
rs_decoder_2	17.947	17.417	17.654	17.654	17.654	17.696	17.696	17.738	17.738	17.738
paj_framebuftop_hierarchy	4.427	3.052	4.083	4.083	4.083	4.139	4.139	4.194	4.194	4.194
s38417	5.076	4.121	4.733	4.733	4.733	4.788	4.788	4.844	4.844	4.844
bigkey	2.762	2.148	2.249	2.249	2.449	2.291	2.291	2.333	2.333	2.533
sig38584.1	4.632	4.513	4.513	4.513	4.573	4.513	4.513	4.513	4.513	4.573
diffeq	4.748	3.339	4.404	4.404	4.404	4.46	4.46	4.515	4.515	4.515
cf_fir_24.16.16	12.136	4.649	11.793	9.754	9.682	11.848	9.754	11.904	9.754	9.737
oc54_cpu	18.126	11.777	17.783	17.783	17.783	17.838	17.838	17.894	17.894	17.894
cf_fir_3.8.8	10.705	4.426	10.361	7.41	7.218	10.417	7.41	10.472	7.41	7.304
diffeq_paj_convert	37.347	37.023	37.041	37.041	37.223	37.068	37.068	37.107	37.107	37.223
Geomean	7.88	5.371	7.454	7.134	7.162	7.513	7.186	7.572	7.238	7.274
Ratio to Critical Path	1	0.682	0.946	0.905	0.909	0.953	0.912	0.961	0.919	0.923
Ratio to PL_{Opt}		1	1.387	1.327	1.333	1.397	1.337	1.409	1.348	1.353

Table 5.1: Achievable Clock period (ns) using flip-flops without any time borrowing (Critical Path), optimal clock period without considering short path constraints (PL_{opt}), pulsed latches heuristic (PL_{heur}), iterative improvement with pulsed latches (PL_{iter}), and clock skew (CS) subject to different minimum delay assumptions.

Chapter 6

Delay Padding

Although the optimization discussed in the previous chapter improves the clock period subject to all short paths, we explore the possibility of increasing the pulse width for more time borrowing opportunities. As this will introduce short path violations that cannot be fixed with flip-flops, we explore fixing these violations by increasing their path delays by taking a more circuitous route in the router. This is known as *delay padding*. Section 6.1 discusses how we select a wider pulse width. As using a larger pulse width will expose violating short paths that cannot be fixed by using flip-flops, Section 6.2 details how we identify such violating short paths. Section 6.3 discusses the two delay padding strategies that were implemented in VPR’s router. Section 6.4 describes the short paths that usually are the hardest to fix, intra-CLB paths. Finally, Section 6.5 presents experimental results on pulsed-latch timing optimization with delay padding.

The process of determining *which* combinational paths to pad for a wider pulse width is a side-effect of our latch-based optimizations. Combinational paths that limit the pulse width are flagged by assigning a minimum delay constraint to nets on limiting combinational paths. This information guides the router to re-route certain nets such that the minimum delay constraints are satisfied. After certain nets are re-routed subject to the constraints, latch-based optimizations are applied again.

6.1 Pulse Width Selection

Reducing the clock period further than what iterative improvement with pulsed latches could achieve requires increasing the pulse width. A violating short path terminating at a sequential element that cannot be converted into a flip-flop due to time borrowing requirements is what usually stopped iterative improvement from improving performance even further. Therefore, using an even larger pulse width will introduce short path violations. Delay padding can fix such short path violations, thereby enabling further improvements.

If we gradually increase the pulse width like the scheme used by iterative improvement, each iteration would require calling the router to fix all the new short path violations introduced by the wider pulse width. Rather than invoking the router every iteration, we select a target pulse width and invoke the router *once* to fix all the violating short paths, as shown in Fig. 6.1. Afterwards, timing optimization using pulsed latches can be invoked to re-calculate the attainable clock period using a single pulse width.

Figure 6.1: Delay padding in relation to rest of CAD flow

The short path delays are once again used to select the target pulse width. Assuming

short path delays are sorted in ascending order, a new short path delay is selected based on the position of the short path delay that determines the current pulse width in the sorted list and a computed offset. This offset is a percentage of the total number of short path delays. For example, suppose there are 100 short paths in a given circuit and that the current pulse width for the circuit is determined by the 10th shortest short path. An offset of 10% implies that the $10 + 100 \times 0.1 = 20$ th shortest short path is used to determine the new pulse width. It is possible that the 20th shortest short path may have the same delay as the 10th, and thus no improvements in pulse width and consequently clock period are possible. As our results will show, this methodology of pulse width selection is at the mercy of the short path delay distributions and the position of the initial short path that determines the pulse width.

6.2 Short Path Identification

The iterative improvement timing optimization discussed in Section 5.2 will maximize the clock period subject to the short path constraints. The algorithm relied on using flip-flops to avoid potential short path violations. However, forcing a flip-flop to be a sequential element can deny long paths terminating at such a flip-flop the ability to borrow time, if needed. The implication of this, as discussed in Section 5.3, is that maximizing the pulse width may not lead to the best achievable clock period because doing so may require forcing too many latches to be flip-flops. But what if we could fix short path violations outright rather than avoiding them with flip-flops? Fixing violations with delay padding would avoid the necessity of forcing certain sequential elements to be flip-flops. Without flip-flops denying the possibility of time borrowing in certain areas of the circuit, maximizing the pulse width would then be directly correlated to clock period reduction. Using delay padding as a method to combat short path violations, we would like to increase the pulse width beyond what iterative improvement could achieve for further

performance gains. This certainly will introduce short path violations that need to be padded.

Before we can fix the short path violations, they must be identified first. One naive approach is to target *all* short paths whose delay are less than the target pulse width for delay padding. One can imagine that the number of paths that need to be fixed can quickly grow out of hand. Rather than padding all such short paths, Algorithm 6 shows how only a subset of such short paths need to be targeted for delay padding.

Algorithm 6 Short Path Identification

Input: W_{target} , G , E_{min} , P_{opt} , ω_{opt}

Output: α

```

1: for  $e(u, v) \in \text{sorted } E$  do
2:    $d_{min}(u, v) \leftarrow$  short path delay of  $e(u, v)$  from  $E_{min}$ 
3:   if  $T_{cq} + d_{min}(u, v) < W_{target} + T_h$  then
4:     latchRequired  $\leftarrow$  FALSE
5:     for  $f(u, v) \in fanin(v)$  do
6:        $l_u \leftarrow \min(\omega_{opt}[u], W_{target})$ 
7:       if  $l_u + T_{dq} + d_{max}(u, v) + T_{su} > P_{opt}$  then
8:         latchRequired  $\leftarrow$  TRUE
9:       break
10:    end if
11:  end for
12:  if latchRequired then
13:     $\alpha \leftarrow \alpha \cup e(u, v)$ 
14:  end if
15: end if
16: end for

```

The inputs to the algorithm are the new target pulse width, W_{target} , the circuit represented in graph form, G , all short paths in G , E_{min} , the optimal clock period P_{opt} without considering any short paths and the set of latencies needed to implement P_{opt} , ω_{opt} . The output, α , is the set of paths that require delay padding. The algorithm iterates over the short paths in ascending order, shown at line (1). Line (3) checks whether or not a short path with delay $d_{min}(u, v)$ would cause a hold-time violation.

If the check at line (3) evaluates to true, the naive approach would immediately classify it as a path that requires delay padding. This implicitly assumes that the short

path terminates at a latch. We can avoid fixing such a path if we can determine that latch-based timing optimization will force it to be a flip-flop after delay padding. Because forcing an element to be a flip-flop can deny time borrowing opportunities for long paths, we must ensure that no long path terminating at the sequential element actually needs to borrow time. Since we do not know the achievable clock period with W_{target} yet, we use P_{opt} and ω_{opt} to test if the sequential element can be a flip-flop. This is conservative because P_{opt} is a lower bound on the achievable clock period, and as a result, ω_{opt} will never underestimate the time borrowing requirements at any sequential element. The loop at lines (5)-(11) uses this property to prune away paths that terminate at sequential elements that do not require time borrowing. Line (6) shows that we can be smarter about when a long path launches from the source latch, u . We know that at most, no sequential element can borrow more time than W_{target} . Furthermore, ω_{opt} provides an upper bound on the amount of time u needs to borrow. Using these two bounds, we can set the starting latency or launch time at u , l_u , to be the minimum of these two values. Using l_u , lines (7)-(10) checks whether or not the long path terminating at v will require time borrowing. If one such path exists, v must be a latch. Therefore, this short path can only be fixed by delay padding, and is added to α at line (13).

Using P_{opt} and ω_{opt} to predict whether or not a sequential element can be a flip-flop relies on the fact that delay padding does not change long path delays between sequential elements. It is possible that the change in the delay of some long path can force a sequential element to remain a latch, even though path identification predicted that it can be a flip-flop. This scenario will expose short paths that should have been delay padded, but were not. The presence of a single such short path can prevent increases in pulse width after delay padding. Hence, the approach used to determine which paths to lengthen is a heuristic.

6.3 Delay Padding Strategies

We experimented with two different delay padding strategies:

1. Minimally disruptive modifications to an already routed design by using only free routing resources
2. Complete rip-up and re-route with minimum delay constraints on certain paths

The obvious advantage of minimally disruptive delay padding is that the probability that delay padding can alter long path delays is greatly reduced. The downside is that the search space for maze expansion is reduced if only free routing resources can be used. This leads to the possibility that some minimum delay constraints can never be satisfied. Before minimally disruptive routing can occur, the existing route to a pin with a minimum delay constraint must be ripped up. Fig. 6.2 shows two different rip-up scenarios.

Figure 6.2: Different route rip-up scenarios for minimally disruptive delay padding

Suppose pins t_1 , t_2 , and t_4 in Fig. 6.2(a) have minimum delay constraints. The dashed lines show what gets ripped up. Essentially, everything except routing that is necessary to reach t_3 is ripped up. Now suppose t_3 has a minimum delay constraint rather than t_1 , as shown in Fig. 6.2(b). The consequence of this is everything up to the fork leading to t_1 gets ripped up. In contrast, complete rip-up and re-route provides a larger search space to meet minimum delay constraints, but such routes can more likely increase long path delays.

Fig. 6.3 shows how both strategies actually find a route between source and target subject to a minimum delay constraint.

Figure 6.3: Delay padding flowchart

Central to both strategies is maze expansion, with a twist. Conventional FPGA routing, as described in Section 2.2.2, terminates maze expansion after the search wavefront originating from the source reaches the target. At this point, the priority queue used to direct the maze expansion still contains unexplored nodes that potentially may be part of a longer path between the source and target. We exploit this property to iteratively find alternate paths to the target, until the minimum delay constraint is met. A couple of corner cases are necessary in case a minimum delay constraint cannot be satisfied. If the target cannot be reached at some iteration of repeated maze expansion, we restore the best path found so far. If no path has been found already, we declare the net to be

unrouteable.

6.4 Intra-CLB Paths

A big hurdle to delay padding is the presence of extremely short combinational paths that start and terminate at sequential elements belonging to the same CLB. Recall from Chapter 3 that the hold-time constraint is defined as:

$$T_{cq} + cd_{ji} \geq W_i + T_h, \forall j \rightarrow i \quad (6.1)$$

Since T_{cq} and T_h are properties of a latch, it quickly becomes obvious that W_i , the amount of time borrowing allowed at latch i depends on the delay of short paths, cd_{ji} , terminating at i . Since setting $W_i = 0$ would yield the standard flip-flop hold-time constraint and any non-zero value for W_i would make the hold-time constraint harder to satisfy. The fast interconnect inside a CLB that is designed to reduce long path delays actually hinders a pulsed latch's ability to borrow time.

A subset of these short intra-CLB paths are paths that start and end at the same flip-flop/latch. We refer to these paths as *self-loops*. Self-loops are interesting because they are far more susceptible to the latch hold-time constraint given in inequality (6.1) than the clock skew hold-time constraint when using flip-flops only. To understand why, we repeat the clock skew hold-time constraint given in Section 2.3:

$$D_j + T_{cq} + cd_{ji} \geq D_i + T_{hold}, \forall j \rightarrow i \quad (6.2)$$

Recall that D_j and D_i represent the delays on the clock line, or latencies. A self-loop implies $i = j$ and therefore $D_j = D_i$, resulting in the standard flip-flop hold-time constraint, which is easier to satisfy than the latch hold-time constraint. The obvious method to fixing intra-CLB short path violations is to force the path to use external routing resources

and re-enter the CLB using an input pin. This requires an additional CLB input pin and output pin if there is no external fanout to other CLBs, and additional routing resources. Such requirements made some minimum delay constraints impossible to satisfy when attempting delay padding with free routing resources due to routing blockages.

6.5 Experimental Study

We explore the possibility of reducing the clock period even further by fixing short path violations with delay padding. The results of the two delay padding strategies, delay padding using only free routing resources (FRR) and complete rip-up and re-route with delay padding (CR) are presented here. For conciseness, the results presented here only use 70% of VPR’s minimum path delays. We did not observe any abnormal side effects with different short path delay scaling factors. Otherwise, we performed the experiments presented in this section with the same assumptions first discussed in Section 5.4:

- The FPGA logic architecture used a LUT size of 6, and cluster size of 8.
- The routing architecture used 50% length-4 and 50% length-2 routing segments.
- Each benchmark was given an extra 30% in channel capacity on top of the minimum required for routing, in order to emulate a “medium stress” scenario.
- The results for each circuit were averaged over four runs, each using a different placement seed value.
- Intrinsic latch and flip-flop timing parameters were not altered.

One important assumption with respect to the flip-flop timing parameters we have selected is that short paths terminating at a flip-flop *cannot* cause a short path violation. Recent research has shown that this assumption is *not* valid for state-of-the-art FPGAs [20]. We believe this was a necessary assumption because VPR 5.0 does not model short paths

in the first place and therefore, any attempt to model them accurately would require more realistic delay and architectural modeling. However, flip-flop short path violations are certainly easier to fix than ones caused by latches and our work on avoiding short path violations using flip-flops will still be applicable.

To ensure an increase in pulse width is not hindered by the inability to fix intra-CLB short paths due to lack of input pins, two free input pins of every CLB were reserved for their use¹. Although this modification caused differences in the absolute numbers when compared to the results presented in Section 5.4, we did not notice any differences in the relative gains.

Table 6.1 shows the results of FRR-style and CR-style delay padding with 5%, 10%, and 15% pulse width offsets. The gains of these two delay padding strategies are compared against a benchmark’s conventional “Critical Path” and gains achieved with pulse latches using the iterative improvement algorithm, “ PL_{iter} ”. Benchmarks `sv_chip1_hierarchy`, `des_area`, `s38584.1`, and `cf_fir_24.16.16` are not shown because they either can achieve the optimal clock period with pulsed latches or none of the pulse width offsets could actually increase the target pulse width. Cells marked by an “x” saw no improvements with a specific offset because it could not increase the target pulse width. Cells marked by an “u” were not routeable either due to routing blockages when using FRR-style delay padding or unresolveable routing congestion. Because these results are averaged over four runs with different seeds, all four seeds of `paj_raygentop_hierarchy` either saw no improvement or were unrouteable under all scenarios, which is why every cell is marked “xu”.

Results are aggregated for benchmarks that successfully routed under all scenarios. Benchmarks that showed no improvement or were unrouteable were not included because it disrupts the ability to compare results across all scenarios in a valid manner. The most promising result is that improvements to PL_{iter} can be made under all offset scenarios for both styles of delay padding. The best scenarios involved using CR-style delay padding

¹Output CLB pins are automatically reserved for BLEs that need them in VPR 5.0.

with a 10% or 15% pulse width offset. These offsets increased overall circuit performance by 6.6%, attained an additional 2.4% on top of iterative improvement and was also 1-2% better than FRR-style delay padding for their respective offsets.

Pulse Width Offset			5%		10%		15%	
Clock Period (ns)	Critical Path	PL_{Iter}	FRR	CR	FRR	CR	FRR	CR
clma	7.528	7.241	7.028	6.971	7.017	7.093	7.027	6.942
fir_scu_rtl_restructured	15.254	14.966	14.649	14.556	14.631	14.408	14.631	14.229
des_perf	3.348	2.89	2.89	2.757	2.856	2.701	2.856	2.939
iir	15.975	15.256	15.03	15.082	14.953	14.997	14.886	14.874
elliptic	4.449	4.161	4.167	4.083	3.998	3.93	4.215	3.713
rs_decoder_2	17.968	17.768	17.904	17.893	18.124	17.978	17.981	17.892
oc54_cpu	18.181	17.893	17.326	17.303	17.31	17.098	17.31	17.119
diff_eq_paj_convert	37.312	37.025	36.925	36.962	38.572	38.614	39.115	39.179
Geomean	11.623	11.121	10.994	10.885	10.986	10.858	11.063	10.852
Ratio to Critical Path	1	0.957	0.946	0.937	0.945	0.934	0.952	0.934
Ratio to PL_{Iter}		1	0.989	0.979	0.988	0.976	0.995	0.976
Ratio of CR to FRR			1	0.99	1	0.988	1	0.981
s38417	4.874	4.587	4.923	4.681	4.973	5.034	u	5.443
diff_eq	4.717	4.43	u	4.474	u	5.45	u	u
mac1	12.181	11.894	12.537	12.526	13.968	16.877	18.092	u
paj_boundtop_hierarchy	5.868	5.58	6.283	6.699	u	10.557	u	u
cf_fir_3_8_8	10.665	7.152	11.048	u	10.866	u	u	u
sv_chip0_hierarchy	4.147	3.859	x	x	3.87	3.829	3.905	3.886
tseng	4.817	4.529	x	x	4.394	4.6	4.405	4.562
iir1	15.353	15.065	x	x	14.913	15.001	15.068	15.301
rs_decoder_1	14.059	13.771	x	x	13.616	13.775	13.616	14.008
cf_cordic_v_18_18_18	5.113	4.825	x	x	4.561	4.526	4.594	4.87
paj_framebuftop_hierarchy	4.43	4.141	x	x	4.152	4.741	4.088	4.9
bigkey	2.724	2.236	x	x	x	x	2.401	2.41
cf_cordic_v_8_8_8	3.014	2.726	x	x	x	x	2.622	2.415
paj_raygentop_hierarchy	11.197	8.144	xu	xu	xu	xu	xu	xu

Table 6.1: Clock period reduction under different pulse width offset assumptions by delay padding.

For benchmarks that showed no improvement using certain pulse width offsets, large improvements were seen in some cases. For example, a 11.4% improvement over PL_{iter} was achieved for cf_cordic_v_8_8_8 with a 15% offset using CR-style delay padding! However, increases in clock period over PL_{iter} were fairly common also. In particular, any benchmark that was unroutable with some offset yielded clock period gains with other offsets that led to a routable design. These benchmarks either required fixing a lot of short paths violation and/or required a lot of delay padding for some short paths. Not surprisingly, such cases can account for why certain benchmarks saw large increases in the clock period after delay padding. They are evident when we plot the increase in total wirelength required for delay padding versus the change in clock period, as shown in Fig. 6.4.

(a)

(b)

Figure 6.4: Performance gains in relation to additional wirelength necessary to fix short path violations

Fig. 6.4 also contrasts how each delay padding strategy performed across the suite of benchmarks. By inspection, we observe that FRR-style delay padding shown in Fig. 6.4(a) appears to be more consistent in improving performance. The most obvious explanation for this is that FRR is less likely to alter long path delays between sequential elements. Therefore, the probability that short path identification may skip a path that actually requires delay padding is smaller.

So if complete rip-up and re-route can alter long path delays to the point of affecting our ability to correctly identify short path violations, why cannot we fix all short path violations subject to some target pulse width? Table 6.2 contrasts the number of paths that require delay padding with and without the use flip-flops to block short path violations wherever possible using 5%, 10%, and 15% pulse width offsets. The “FixAll” column lists the total number of short paths that require padding to satisfy a certain pulse width offset, whereas the “UseFFs” column gives the number of fixes necessary if flip-flops can be used to block short path violations. Benchmarks that did not require any delay padding for some offsets were not factored into the aggregated results.

Table 6.2 shows that circuits that were least affected by the use of flip-flops to fix potential short path violations were likely to run into routeability issues. Specifically, using flip-flops didn’t do much in reducing the number of paths that required delay padding for `cf_fir_3_8_8`. Not surprisingly, it was unroutable with CR-style delay padding and showed noticeable degradation in performance with FRR-style delay padding. Other benchmarks such as `diffeq` and `paj_raygentop_hierarchy` show similar behaviour. On the other hand, benchmarks that benefit greatly with flip-flop usage such as `clma`, `fir_scu_rtl_restructured`, and `iir` were routable under every scenario and also yielded reductions in the clock period under every scenario. On average, Table 6.2 shows that only 8-12% of the total number of short paths need to be fixed to satisfy a target pulse width. The effect of using flip-flops to avoid short path violations diminishes for certain benchmarks as pulse width offset increases. Two such benchmarks include `des_perf` and `diffeq`. This suggests that for

Pulse Width Offset	5%		10%		15%	
Short Path Violation Fix Method	FixAll	UseFFs	FixAll	UseFFs	FixAll	UseFFs
paj_boundtop_hierarchy	3805	414	7726	1155	11585	2200
sv_chip0_hierarchy	13082	56	14185	78	17023	178
clma	67	2	117	5	167	6
fir_scu_rtl_restructured	407	26	731	62	1043	86
des_perf	563	76	1131	243	1699	444
iir	613	9	893	26	1168	59
paj_raygentop_hierarchy	15535	3819	16539	5171	17543	6525
mac1	985	312	1973	733	2959	1062
elliptic	303	36	480	37	715	47
rs_decoder_2	787	49	1441	83	2137	128
s38417	1628	153	3211	360	4809	681
diffeq	563	115	1105	424	1652	819
oc54_cpu	1086	31	2119	96	3155	172
cf_fir_3_8_8	1132	939	1230	1062	1336	1189
diffeq_paj_convert	902	53	1752	257	2607	489
Geomean	1038	83	1722	182	2390	297
Ratio to FixAll	1	0.08	1	0.106	1	0.124
tseng	0	0	604	62	886	116
iir1	0	0	294	48	443	79
cf_cordic_v_8_8_8	0	0	0	0	619	197
rs_decoder_1	0	0	937	42	1390	110
cf_cordic_v_18_18_18	0	0	3372	259	5015	666
paj_framebuftop_hierarchy	0	0	859	88	1030	79
bigkey	0	0	0	0	489	347

Table 6.2: Comparing the number short paths requiring delay padding with and without the use of using flip-flops.

certain benchmarks, longer short paths are more likely to terminate at sequential elements that require time borrowing. Therefore, delay padding is necessary to fix the potential violation.

Using a fixed pulse width offset for all benchmarks clearly doesn't produce consistent reductions in the clock period. This problem is very much correlated with the ability to avoid fixing certain short path violations with flip-flops as Table 6.2 shows. Pulse width selection strategies that adapt to the short path delays on a per benchmark basis could possibly fare much better.

Chapter 7

Conclusions and Future Work

This thesis explored using pulsed latches for timing optimization purposes. We devised algorithms that selectively insert latches into already-routed flip-flop-based designs for better performance, essentially for “free” performance improvement in FPGAs. Our iterative improvement algorithm using pulsed latches was able to reduce the clock period by approximately 9% with 70% minimum delay assumptions. More importantly, the gains achieved by this algorithm could match clock skew’s gains for most benchmarks. We found this very surprising as clock skew’s hold-time constraint is easier to satisfy. We showed that clock skew may achieve better results than pulsed latches under certain circumstances, but high fan-in and fan-out coupled with imbalanced long and short path combinational delays can nullify clock skew’s advantage quickly.

Despite this surprising result, we observed that short path constraints limited iterative improvement and clock skew’s gains. A 32% reduction in the clock period was possible if short path constraints were not considered. Increasing the delay of certain short paths was explored so that a larger pulse width could be used for more time borrowing opportunities. We experimented with two different delay padding strategies: delay padding using free routing resources only and complete rip-up and re-route. Complete rip-up and re-route achieved better overall results by reducing iterative improvement’s clock period by an

additional 2%. However, delay padding using free resources was more consistent in delivering gains on top of iterative improvement. The major problem with both methods was that the performance of certain benchmarks got worse or were unrouteable due to the extra routing resources needed by delay padding. We believe delay padding could be more effective in two ways:

- A pulse width selection scheme that adapts to a benchmark’s short path delay distribution.
- The delay padding strategy should be a hybrid of the two separate strategies we attempted. That is, use free routing resources whenever possible and only rip-up and re-route what is necessary if routing blockages are present.

So far, we have only discussed router modifications to fix short paths. We wondered whether improved results could be achieved if the place-and-route tools were aware of the time borrowing properties of latches? VPR gauges the criticality of a connection using its *slack ratio*, which is the ratio of the worst-case path delay through a connection, to the current critical path delay. We conducted a preliminary investigation where we altered VPR’s criticality notion to drive place and route using *cycle-slacks* [2, 63]. The cycle-slack ratio of a connection is the cycle ratio of any cycle that uses the connection, to the current MCR of the design. Its definition is thus analogous to slack ratio, making it straightforward to integrate into VPR. Fig. 7.1 shows the benefits of pulsed latch optimization with place and route using cycle-slacks, and our optimization with regular place and route, normalized to 1.

If we ignore short paths completely, we observe a 3% reduction in the MCR when VPR is driven with cycle-slacks. However, with minimum delay constraints, our results indicate an increase in the MCR – i.e. slightly worse results than if VPR is driven by normal slacks! We believe this is a consequence of cycle-slacks being unaware of short paths, which constrain the amount of time borrowing allowed. While cycle-slacks indeed yield

Figure 7.1: Comparing the benefits of pulsed latches with and without using cycle-slacks in VPR

higher performance when short paths are not considered, higher performance requires more time borrowing to be realized, and is therefore more susceptible to short path violations. We noticed that VPR had to deal with many near critical cycles throughout the flow, and since it is difficult to equally optimize all equally-critical cycles, cycle-slacks may drive VPR to optimize the wrong cycles. A direction for future work is to combine cycle-slack-driven FPGA placement with delay padding in routing to hand hold-time violations.

Bibliography

- [1] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Trans. Very Large Scale Integr. Syst.*, 12:288–298, March 2004.
- [2] Christoph Albrecht, Bernhard Korte, Jürgen Schietke, and Jens Vygen. Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip. *Discrete Appl. Math.*, 123:103–127, November 2002.
- [3] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3 GHz fifth generation SPARC64 microprocessor. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 246 – 491 vol.1, 2003.
- [4] Vaughn Betz and Jonathan Rose. VPR: a new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, volume 1304 of *Lecture Notes in Computer Science*, pages 213–222. Springer Berlin / Heidelberg, 1997.
- [5] Vaughn Betz and Jonathan Rose. FPGA routing architecture: segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, FPGA '99*, pages 59–68, New York, NY, USA, 1999. ACM.

- [6] Huimin Bian, Andrew C. Ling, Alexander Choong, and Jianwen Zhu. Towards scalable placement for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 147–156, 2010.
- [7] Jonas Casanova and Jordi Cortadella. Multi-level clustering for clock skew optimization. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 547–554, New York, NY, USA, 2009. ACM.
- [8] Deming Chen, Jason Cong, and Peichen Pan. FPGA Design Automation: A Survey. *Found. Trends Electron. Des. Autom.*, 1:139–169, January 2006.
- [9] L.T. Clark, E.J. Hoffman, J. Miller, M. Biyani, Luyun Liao, S. Strazdus, M. Morrow, K.E. Velarde, and M.A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *Solid-State Circuits, IEEE Journal of*, 36(11):1599 –1608, nov 2001.
- [10] Jean Cochet-terrasson, Guy Cohen, Stephane Gaubert, Michael Mc Gettrick, and Jean pierre Quadrat. Numerical computation of spectral elements in max-plus algebra, 1998.
- [11] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34:171–210, June 2002.
- [12] Jason Cong and Chang Wu. FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 644–649, New York, NY, USA, 1997. ACM.
- [13] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9:385–418, October 2004.
- [14] Vijay Degalahal and Tim Tuan. Methodology for high level estimation of FPGA power consumption. In *Proceedings of the 2005 Asia and South Pacific Design*

- Automation Conference*, ASP-DAC '05, pages 657–660, New York, NY, USA, 2005. ACM.
- [15] Rahul Deokar and Sachin Sapatnekar. A graph-theoretic approach to clock skew optimization. In *In Proc. International Symp. on Circuits and Systems*, pages 407–410, 1994.
- [16] Rahul B. Deokar and Sachin S. Sapatnekar. A fresh look at retiming via clock skew optimization. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 310–315, New York, NY, USA, 1995. ACM.
- [17] Xiao Dong and G.G.F. Lemieux. PGR: Period and glitch reduction via clock skew scheduling, delay padding and GlitchLess. In *IEEE FPT*, December 2009.
- [18] John P. Fishburn. Clock skew optimization. *IEEE Trans. Comput.*, 39:945–951, July 1990.
- [19] Ross H. Freeman. Configurable electrical circuit having configurable logic elements and configurable interconnects, 09 1989. US Patent 07158011.
- [20] R. Fung, V. Betz, and W. Chow. Slack allocation and routing to improve FPGA timing while repairing short-path violations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(4):686–697, april 2008.
- [21] Robert B. Hitchcock, Gordon L. Smith, and David D. Cheng. Timing analysis of computer hardware. *IBM J. Res. Dev.*, 26:100–105, January 1982.
- [22] Yu Hu, Yan Lin, Lei He, and Tim Tuan. Simultaneous time slack budgeting and retiming for dual-vdd FPGA power reduction. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 478–483, New York, NY, USA, 2006. ACM.

- [23] Alexander T. Ishii, Charles E. Leiserson, and Marios C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *J. ACM*, 44:148–199, January 1997.
- [24] Richard M. Karp and James B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics*, 3(1):37 – 45, 1981.
- [25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [26] J.M. Kleinhans, G. Sigl, F.M. Johannes, and K.J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(3):356 –365, mar 1991.
- [27] P.M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill advanced computer science series. Hemisphere Pub. Corp., 1981.
- [28] S. Kozu, M. Daito, Y. Sugiyama, H. Suzuki, H. Morita, M. Nomura, K. Nadehara, S. Ishibuchi, M. Tokuda, Y. Inoue, T. Nakayama, H. Harigai, and Y. Yano. A 100 MHz, 0.4 W RISC processor with 200 MHz multiply adder, using pulse-register technique. In *Solid-State Circuits Conference, 1996. Digest of Technical Papers. 42nd ISSCC., 1996 IEEE International*, pages 140 –141, 432, feb 1996.
- [29] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.
- [30] N.A. Kurd, J.S. Barkarullah, R.O. Dizon, T.D. Fletcher, and P.D. Madland. A multigigahertz clocking scheme for the Pentium(R) 4 microprocessor. *Solid-State Circuits, IEEE Journal of*, 36(11):1647 –1653, nov 2001.

- [31] K.N. Lalgudi and M.C. Papaefthymiou. Fixed-phase retiming for low power design. In *Low Power Electronics and Design, 1996., International Symposium on*, pages 259–264, aug 1996.
- [32] C. Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, sept. 1961.
- [33] Hyein Lee, Seungwhun Paik, and Youngsoo Shin. Pulse width allocation and clock skew scheduling: Optimizing sequential circuits based on pulsed latches. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):355–366, March 2010.
- [34] Seonggwon Lee, Seungwhun Paik, and Youngsoo Shin. Retiming and time borrowing: optimizing high-performance pulsed-latch-based circuits. In *Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD '09*, pages 375–380, New York, NY, USA, 2009. ACM.
- [35] Charles Leiserson and James Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [36] David Lewis and David Cashman. Configurable time borrowing flip-flops, 09 2009. US Patent 7583103.
- [37] Chuan Lin and Hai Zhou. Clock skew scheduling with delay padding for prescribed skew domains. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference, ASP-DAC '07*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.
- [38] B. Lockyear and C. Ebeling. Optimal retiming of level-clocked circuits using symmetric clock schedules. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(9):1097–1109, sep 1994.

- [39] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, Kenneth Kent, and Jonathan Rose. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *ACM Trans. Reconfigurable Technol. Syst.*, 4:32:1–32:23, December 2011.
- [40] N. Maheshwari and S.S. Sapatnekar. Minimum area retiming with equivalent initial states. In *Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on*, pages 216 –219, nov 1997.
- [41] Larry McMurchie and Carl Ebeling. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, FPGA '95, pages 111–117, New York, NY, USA, 1995. ACM.
- [42] Maher N. Mneimneh, Karem A. Sakallah, and John Moondanos. Preserving synchronizing sequences of sequential circuits after retiming. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 579–584, Piscataway, NJ, USA, 2004. IEEE Press.
- [43] José Monteiro, Srinivas Devadas, and Abhijit Ghosh. Retiming sequential circuits for low power. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 398–402, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [44] S.D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T.J. Sullivan, and T. Grutkowski. The implementation of the Itanium 2 microprocessor. *Solid-State Circuits, IEEE Journal of*, 37(11):1448 – 1460, nov 2002.
- [45] Min Ni and Seda Ogrenci Memik. A fast heuristic algorithm for multidomain clock skew scheduling. *IEEE Trans. Very Large Scale Integr. Syst.*, 18:630–637, April 2010.

- [46] S. Paik, S. Lee, and Y. Shin. Retiming pulsed-latch circuits with regulating pulse width. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(8):1114–1127, aug. 2011.
- [47] Marios C. Papaefthymiou and Keith H. Randall. Tim: a timing package for two-phase, level-clocked circuitry. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 497–502, New York, NY, USA, 1993. ACM.
- [48] Kaushik Ravindran, Andreas Kuehlmann, and Ellen Sentovich. Multi-domain clock skew scheduling. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD '03, pages 801–, Washington, DC, USA, 2003. IEEE Computer Society.
- [49] J. Rose, R.J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *Solid-State Circuits, IEEE Journal of*, 25(5):1217–1225, oct 1990.
- [50] K.A. Sakallah, T.N. Mudge, and O.A. Olukotun. CheckTc and minTc: timing verification and optimal clocking of synchronous digital circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 552–555, nov 1990.
- [51] Karem A. Sakallah, Trevor N. Mudge, and Oyekunle A. Olukotun. Analysis and design of latch-controlled synchronous digital circuits. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 111–117, New York, NY, USA, 1990. ACM.
- [52] Sachin Sapatnekar. *Timing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

- [53] S.S. Sapatnekar and R.B. Deokar. Utilizing the retiming-skew equivalence in a practical algorithm for retiming large circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(10):1237 –1248, oct 1996.
- [54] A. Scherer, M. Golden, N. Juffa, S. Meier, S. Oberman, H. Partovi, and F. Weber. An out-of-order three-way superscalar multimedia floating-point unit. In *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, pages 94 –95, 1999.
- [55] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *Solid-State Circuits, IEEE Journal of*, 20(2):510 – 522, apr 1985.
- [56] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic power consumption in Virtex-II FPGA family. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 157–164, New York, NY, USA, 2002. ACM.
- [57] Mike Sheng and Jonathan Rose. Mixing buffers and pass transistors in fpga routing architectures. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 75–84, New York, NY, USA, 2001. ACM.
- [58] Narendra Shenoy, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Graph algorithms for clock schedule optimization. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, ICCAD '92, pages 132–136, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [59] Narendra Shenoy and Richard Rudell. Efficient implementation of retiming. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 226–233, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

- [60] Narendra V. Shenoy, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Retiming of circuits with single phase transparent latches. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, ICCD '91, pages 86–89, Washington, DC, USA, 1991. IEEE Computer Society.
- [61] M. Shoji. Elimination of process-dependent clock skew in CMOS VLSI. *Solid-State Circuits, IEEE Journal of*, 21(5):875 – 880, oct 1986.
- [62] Deshanand P. Singh and Stephen D. Brown. Constrained clock shifting for field programmable gate arrays. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 121–126, New York, NY, USA, 2002. ACM.
- [63] Deshanand P. Singh and Stephen D. Brown. Integrated retiming and placement for field programmable gate arrays. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 67–76, New York, NY, USA, 2002. ACM.
- [64] Deshanand P. Singh, Valavan Manohararajah, and Stephen D. Brown. Incremental retiming for FPGA physical synthesis. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 433–438, New York, NY, USA, 2005. ACM.
- [65] V. Singhal, S. Malik, and R.K. Brayton. The case for retiming with explicit reset circuitry. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 618 –625, nov 1996.
- [66] T. G. Szymanski. Computing optimal clock schedules. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 399–404, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

- [67] Baris Taskin and Ivan S. Kourtev. Delay insertion method in clock skew scheduling. In *Proceedings of the 2005 international symposium on Physical design*, ISPD '05, pages 47–54, New York, NY, USA, 2005. ACM.
- [68] Bill Teng and Jason H. Anderson. Latch-based performance optimization for FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 58–63, Washington, DC, USA, 2011. IEEE Computer Society.
- [69] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *The Journal of VLSI Signal Processing*, 28:7–27, 2001.
- [70] H.J. Touati and R.K. Brayton. Computing the initial states of retimed circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(1):157 –162, jan 1993.
- [71] R.-S. Tsay. An exact zero-skew clock routing algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(2):242 –249, feb 1993.
- [72] A. van der Werf, J. L. Van Meerbergen, E. H. L. Aarts, W. F. J. Verhaegh, and P. E. R. Lippens. Efficient timing constraint derivation for optimal retiming high speed processing units. In *Proceedings of the 7th international symposium on High-level synthesis*, ISSS '94, pages 48–53, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [73] N. Viswanathan and C.C.-N. Chu. FastPlace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(5):722 – 733, may 2005.
- [74] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003*

- ACM/SIGDA eleventh international symposium on Field programmable gate arrays, FPGA '03*, pages 185–194, New York, NY, USA, 2003. ACM.
- [75] U. Weinmann and W. Rosenstiel. Technology mapping for sequential circuits based on retiming techniques. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, pages 318–323, sep 1993.
- [76] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Trans. Embed. Comput. Syst.*, 3:534–574, August 2004.
- [77] Xilinx, Inc., San Jose, CA. *Virtex-6 FPGA Configurable Logic Block*, 2009.
- [78] Xilinx, Inc., San Jose, CA. *Virtex-6 FPGA Clocking Resources*, 2011.
- [79] Chao-Yang Yeh and Malgorzata Marek-Sadowska. Skew-programmable clock design for FPGA and skew-aware placement. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, FPGA '05*, pages 33–40, New York, NY, USA, 2005. ACM.
- [80] Neal E. Young, Robert E. Tarjan, and James B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.
- [81] Jihan Zhu and Peter Sutton. FPGA implementations of neural networks a survey of a decade of progress. In Peter Y. K. Cheung and George Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 1062–1066. Springer Berlin / Heidelberg, 2003.